

The UNITY Formalism

Sidi Ehmety and Lawrence C. Paulson

September 11, 2023

Contents

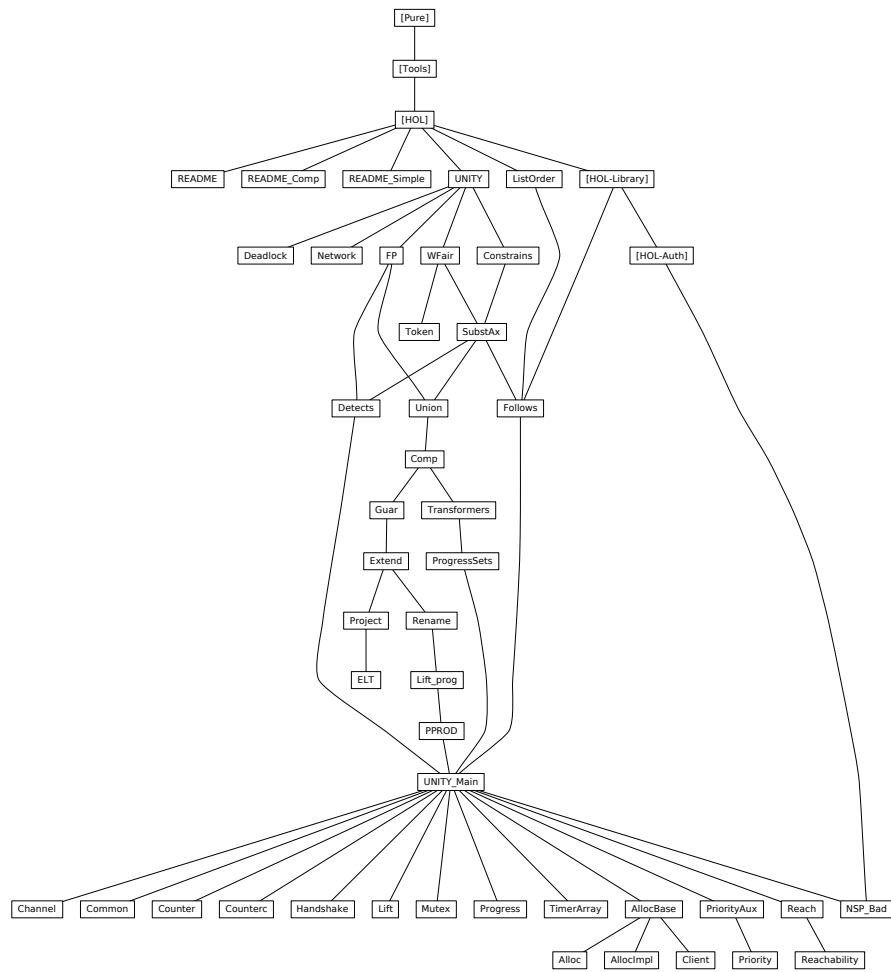
1	The Basic UNITY Theory	2
1.0.1	The abstract type of programs	3
1.0.2	Inspectors for type "program"	3
1.0.3	Equality for UNITY programs	3
1.0.4	co	4
1.0.5	Union	4
1.0.6	Intersection	5
1.0.7	unless	5
1.0.8	stable	5
1.0.9	Union	6
1.0.10	Intersection	6
1.0.11	invariant	6
1.0.12	increasing	6
1.0.13	Theoretical Results from Section 6	7
1.0.14	Ad-hoc set-theory rules	7
1.1	Partial versus Total Transitions	7
1.1.1	Basic properties	8
1.2	Rules for Lazy Definition Expansion	9
1.2.1	Inspectors for type "program"	9
2	Fixed Point of a Program	10
3	Progress	10
3.1	transient	11
3.2	ensures	12
3.3	leadsTo	13
3.4	PSP: Progress-Safety-Progress	16
3.5	Proving the induction rules	16
3.6	wlt	17
3.7	Completion: Binary and General Finite versions	18
4	Weak Safety	19
4.1	traces and reachable	19
4.2	Co	20
4.3	Stable	21
4.4	Increasing	22
4.5	The Elimination Theorem	22

4.6	Specialized laws for handling Always	23
4.7	"Co" rules involving Always	23
4.8	Totalize	24
5	Weak Progress	25
5.1	Specialized laws for handling invariants	25
5.2	Introduction rules: Basis, Trans, Union	25
5.3	Derived rules	26
5.4	PSP: Progress-Safety-Progress	28
5.5	Induction rules	29
5.6	Completion: Binary and General Finite versions	30
6	The Detects Relation	31
7	Unions of Programs	32
7.1	SKIP	32
7.2	SKIP and safety properties	33
7.3	Join	33
7.4	JN	33
7.5	Algebraic laws	34
7.6	Laws Governing \sqcup	34
7.7	Safety: co, stable, FP	34
7.8	Progress: transient, ensures	35
7.9	the ok and OK relations	36
7.10	Allowed	37
7.11	<i>safety_prop</i> , for reasoning about given instances of "ok"	38
8	Composition: Basic Primitives	39
8.1	The component relation	39
8.2	The preserves property	41
9	Guarantees Specifications	43
9.1	Existential Properties	44
9.2	Universal Properties	44
9.3	Guarantees	45
9.4	Distributive Laws. Re-Orient to Perform Miniscoping	45
9.5	Guarantees: Additional Laws (by lcp)	46
9.6	Guarantees Laws for Breaking Down the Program (by lcp)	47
10	Extending State Sets	49
10.1	Restrict	50
10.2	Trivial properties of f, g, h	51
10.3	<i>extend_set</i> : basic properties	52
10.4	<i>project_set</i> : basic properties	52
10.5	More laws	53
10.6	<i>extend_act</i>	53
10.7	extend	54
10.8	Safety: co, stable	56
10.9	Weak safety primitives: Co, Stable	56
10.10	Progress: transient, ensures	58
10.11	Proving the converse takes some doing!	58

10.12	preserves	59
10.13	Guarantees	59
11	Renaming of State Sets	60
11.1	inverse properties	60
11.2	the lattice operations	62
11.3	Strong Safety: <i>co</i> , <i>stable</i>	62
11.4	Weak Safety: <i>Co</i> , <i>Stable</i>	62
11.5	Progress: <i>transient</i> , <i>ensures</i>	63
11.6	"image" versions of the rules, for lifting "guarantees" properties	64
12	Replication of Components	65
12.1	Injectiveness proof	65
12.2	Surjectiveness proof	66
12.3	The Operator <i>lift_set</i>	67
12.4	The Lattice Operations	67
12.5	Safety: <i>constrains</i> , <i>stable</i> , <i>invariant</i>	67
12.6	Progress: <i>transient</i> , <i>ensures</i>	68
12.7	Lemmas to Handle Function Composition (o) More Consistently	70
12.8	More lemmas about <i>extend</i> and <i>project</i>	70
12.9	OK and "lift"	70
13	The Prefix Ordering on Lists	73
13.1	preliminary lemmas	74
13.2	<i>genPrefix</i> is a partial order	74
13.3	recursion equations	75
13.4	The type of lists is partially ordered	76
13.5	<i>pfixLe</i> , <i>pfixGe</i> : properties inherited from the translations	78
14	The Follows Relation of Charpentier and Sivilotte	79
14.1	Destruction rules	80
14.2	Union properties (with the subset ordering)	80
14.3	Multiset union properties (with the multiset ordering)	81
15	Predicate Transformers	82
15.1	Defining the Predicate Transformers <i>wp</i> , <i>awp</i> and <i>wens</i>	82
15.2	Defining the Weakest Ensures Set	84
15.3	Properties Involving Program Union	85
15.4	The Set <i>wens_set F B</i> for a Single-Assignment Program	85
16	Progress Sets	88
16.1	Complete Lattices and the Operator <i>c1</i>	88
16.2	Progress Sets and the Main Lemma	90
16.3	The Progress Set Union Theorem	91
16.4	Some Progress Sets	92
16.4.1	Lattices and Relations	92
16.4.2	Decoupling Theorems	93
16.5	Composition Theorems Based on Monotonicity and Commutativity	93
16.5.1	Commutativity of <i>c1 L</i> and assignment.	93
16.5.2	Commutativity of Functions and Relation	94
16.6	Monotonicity	95

17 Comprehensive UNITY Theory	95
18 The Token Ring	99
18.1 Definitions	99
18.2 Progress under Weak Fairness	100
18.3 Progress	106
19 Analyzing the Needham-Schroeder Public-Key Protocol in UNITY	120
19.1 Inductive Proofs about <i>ns_public</i>	121
19.2 Authenticity properties obtained from NS2	121
19.3 Authenticity properties obtained from NS2	122
20 A Family of Similar Counters: Original Version	125
21 A Family of Similar Counters: Version with Compatibility	126
22 The priority system	130
22.1 Component correctness proofs	131
22.2 System properties	132
22.3 The main result: above set decreases	133
23 Progress Set Examples	135
23.1 The Composition of Two Single-Assignment Programs	135
23.1.1 Calculating <i>wens_set FF {k..}</i>	135
23.1.2 Proving $FF \in UNIV \mapsto \{k..\}$	136
24 Common Declarations for Chandy and Charpentier's Allocator	136
24.1 State definitions. OUTPUT variables are locals	138
24.1.1 Resource allocation system specification	138
24.1.2 Client specification (required)	139
24.1.3 Allocator specification (required)	139
24.1.4 Network specification	140
24.1.5 State mappings	141
24.1.6 bijectivity of <i>sysOfClient</i>	143
24.1.7 bijectivity of <i>client_map</i>	143
24.2 o-simprules for <i>sysOfAlloc</i> [MUST BE AUTOMATED]	144
24.3 o-simprules for <i>sysOfClient</i> [MUST BE AUTOMATED]	144
24.4 Components Lemmas [MUST BE AUTOMATED]	146
24.5 Proof of the safety property (1)	149
24.6 Proof of the progress property (2)	150
25 Implementation of a multiple-client allocator from a single-client allocator	152
25.1 Theorems for Merge	155
25.2 Theorems for Distributor	156
25.3 Theorems for Allocator	157
26 Distributed Resource Management System: the Client	157

<i>CONTENTS</i>	5
27 Projections of State Sets	161
27.1 Safety	161
27.2 "projecting" and union/intersection (no converses)	162
27.3 Reachability and project	164
27.4 Converse results for weak safety: benefits of the argument C . . .	164
27.5 A lot of redundant theorems: all are proved to facilitate reasoning about guarantees.	165
27.6 leadsETo in the precondition (??)	166
27.6.1 transient	166
27.6.2 ensures – a primitive combining progress with safety . . .	166
27.7 Towards the theorem <i>project_Ensures_D</i>	167
27.8 Guarantees	168
27.9 guarantees corollaries	168
27.9.1 Some could be deleted: the required versions are easy to prove	168
27.9.2 Guarantees with a leadsTo postcondition	169
28 Progress Under Allowable Sets	169



1 The Basic UNITY Theory

theory UNITY imports Main begin

definition

```
"Program =
  {(init:: 'a set, acts :: ('a * 'a)set set,
   allowed :: ('a * 'a)set set). Id ∈ acts & Id ∈ allowed}"
```

```
typedef 'a program = "Program :: ('a set * ('a * 'a) set set * ('a * 'a) set
set) set"
```

```
morphisms Rep_Program Abs_Program
⟨proof⟩
```

definition Acts :: "'a program => ('a * 'a)set set" where

```
"Acts F == (%(init, acts, allowed). acts) (Rep_Program F)"
```

definition "constrains" :: "['a set, 'a set] => 'a program set" (infixl "co" 60) where

```
"A co B == {F. ∀act ∈ Acts F. act 'A ⊆ B}"
```

definition unless :: "['a set, 'a set] => 'a program set" (infixl "unless" 60) where

```
"A unless B == (A-B) co (A ∪ B)"
```

definition mk_program :: "('a set * ('a * 'a)set set * ('a * 'a)set set) => 'a program" where

```
"mk_program == (%(init, acts, allowed).
  Abs_Program (init, insert Id acts, insert Id allowed))"
```

definition Init :: "'a program => 'a set" where

```
"Init F == (%(init, acts, allowed). init) (Rep_Program F)"
```

definition AllowedActs :: "'a program => ('a * 'a)set set" where

```
"AllowedActs F == (%(init, acts, allowed). allowed) (Rep_Program F)"
```

definition Allowed :: "'a program => 'a program set" where

```
"Allowed F == {G. Acts G ⊆ AllowedActs F}"
```

definition stable :: "'a set => 'a program set" where

```
"stable A == A co A"
```

definition strongest_rhs :: "['a program, 'a set] => 'a set" where

```
"strongest_rhs F A == ⋂ {B. F ∈ A co B}"
```

definition invariant :: "'a set => 'a program set" where

```
"invariant A == {F. Init F ⊆ A} ∩ stable A"
```

definition increasing :: "['a => 'b::order] => 'a program set" where

```
— Polymorphic in both states and the meaning of ≤
"increasing f == ⋂ z. stable {s. z ≤ f s}"
```

1.0.1 The abstract type of programs

```
lemmas program_typedef =
  Rep_Program Rep_Program_inverse Abs_Program_inverse
  Program_def Init_def Acts_def AllowedActs_def mk_program_def
```

```
lemma Id_in_Acts [iff]: "Id ∈ Acts F"
⟨proof⟩
```

```
lemma insert_Id_Acts [iff]: "insert Id (Acts F) = Acts F"
⟨proof⟩
```

```
lemma Acts_nonempty [simp]: "Acts F ≠ {}"
⟨proof⟩
```

```
lemma Id_in_AllowedActs [iff]: "Id ∈ AllowedActs F"
⟨proof⟩
```

```
lemma insert_Id_AllowedActs [iff]: "insert Id (AllowedActs F) = AllowedActs
F"
⟨proof⟩
```

1.0.2 Inspectors for type "program"

```
lemma Init_eq [simp]: "Init (mk_program (init,acts,allowed)) = init"
⟨proof⟩
```

```
lemma Acts_eq [simp]: "Acts (mk_program (init,acts,allowed)) = insert Id
acts"
⟨proof⟩
```

```
lemma AllowedActs_eq [simp]:
  "AllowedActs (mk_program (init,acts,allowed)) = insert Id allowed"
⟨proof⟩
```

1.0.3 Equality for UNITY programs

```
lemma surjective_mk_program [simp]:
  "mk_program (Init F, Acts F, AllowedActs F) = F"
⟨proof⟩
```

```
lemma program_equalityI:
  "[| Init F = Init G; Acts F = Acts G; AllowedActs F = AllowedActs G |]
  ==> F = G"
⟨proof⟩
```

```
lemma program_equalityE:
  "[| F = G;
  [| Init F = Init G; Acts F = Acts G; AllowedActs F = AllowedActs G
  |]
  ==> P |] ==> P"
⟨proof⟩
```

```
lemma program_equality_iff:
```


"(F=G) =
 (Init F = Init G & Acts F = Acts G & AllowedActs F = AllowedActs G)"
 <proof>

1.0.4 co

lemma constrainsI:
 "(!!act s s'. [| act ∈ Acts F; (s,s') ∈ act; s ∈ A |] ==> s' ∈ A)"
 ==> F ∈ A co A"
 <proof>

lemma constrainsD:
 "[| F ∈ A co A'; act ∈ Acts F; (s,s') ∈ act; s ∈ A |] ==> s' ∈ A'"
 <proof>

lemma constrains_empty [iff]: "F ∈ {} co B"
 <proof>

lemma constrains_empty2 [iff]: "(F ∈ A co {}) = (A={})"
 <proof>

lemma constrains_UNIV [iff]: "(F ∈ UNIV co B) = (B = UNIV)"
 <proof>

lemma constrains_UNIV2 [iff]: "F ∈ A co UNIV"
 <proof>

monotonic in 2nd argument

lemma constrains_weaken_R:
 "[| F ∈ A co A'; A' ≤ B' |] ==> F ∈ A co B'"
 <proof>

anti-monotonic in 1st argument

lemma constrains_weaken_L:
 "[| F ∈ A co A'; B ⊆ A |] ==> F ∈ B co A'"
 <proof>

lemma constrains_weaken:
 "[| F ∈ A co A'; B ⊆ A; A' ≤ B' |] ==> F ∈ B co B'"
 <proof>

1.0.5 Union

lemma constrains_Un:
 "[| F ∈ A co A'; F ∈ B co B' |] ==> F ∈ (A ∪ B) co (A' ∪ B'"
 <proof>

lemma constrains_UN:
 "(!!i. i ∈ I ==> F ∈ (A i) co (A' i))
 ==> F ∈ (⋃ i ∈ I. A i) co (⋃ i ∈ I. A' i)"
 <proof>

lemma constrains_Un_distrib: "(A ∪ B) co C = (A co C) ∩ (B co C)"

<proof>

lemma *constrains_UN_distrib*: " $(\bigcup i \in I. A\ i)\ \text{co}\ B = (\bigcap i \in I. A\ i\ \text{co}\ B)$ "
<proof>

lemma *constrains_Int_distrib*: " $C\ \text{co}\ (A \cap B) = (C\ \text{co}\ A) \cap (C\ \text{co}\ B)$ "
<proof>

lemma *constrains_INT_distrib*: " $A\ \text{co}\ (\bigcap i \in I. B\ i) = (\bigcap i \in I. A\ \text{co}\ B\ i)$ "
<proof>

1.0.6 Intersection

lemma *constrains_Int*:
" $[| F \in A\ \text{co}\ A'; F \in B\ \text{co}\ B' |] \implies F \in (A \cap B)\ \text{co}\ (A' \cap B')$ "
<proof>

lemma *constrains_INT*:
" $(\forall i. i \in I \implies F \in (A\ i)\ \text{co}\ (A'\ i)) \implies F \in (\bigcap i \in I. A\ i)\ \text{co}\ (\bigcap i \in I. A'\ i)$ "
<proof>

lemma *constrains_imp_subset*: " $F \in A\ \text{co}\ A' \implies A \subseteq A'$ "
<proof>

The reasoning is by subsets since "co" refers to single actions only. So this rule isn't that useful.

lemma *constrains_trans*:
" $[| F \in A\ \text{co}\ B; F \in B\ \text{co}\ C |] \implies F \in A\ \text{co}\ C$ "
<proof>

lemma *constrains_cancel*:
" $[| F \in A\ \text{co}\ (A' \cup B); F \in B\ \text{co}\ B' |] \implies F \in A\ \text{co}\ (A' \cup B')$ "
<proof>

1.0.7 unless

lemma *unlessI*: " $F \in (A-B)\ \text{co}\ (A \cup B) \implies F \in A\ \text{unless}\ B$ "
<proof>

lemma *unlessD*: " $F \in A\ \text{unless}\ B \implies F \in (A-B)\ \text{co}\ (A \cup B)$ "
<proof>

1.0.8 stable

lemma *stableI*: " $F \in A\ \text{co}\ A \implies F \in \text{stable}\ A$ "
<proof>

lemma *stableD*: " $F \in \text{stable}\ A \implies F \in A\ \text{co}\ A$ "
<proof>

lemma *stable_UNIV [simp]*: " $\text{stable}\ \text{UNIV} = \text{UNIV}$ "
<proof>

1.0.9 Union

lemma *stable_Un*:

"[| F ∈ stable A; F ∈ stable A' |] ==> F ∈ stable (A ∪ A')"

⟨proof⟩

lemma *stable_UN*:

"(!!i. i ∈ I ==> F ∈ stable (A i)) ==> F ∈ stable (⋃ i ∈ I. A i)"

⟨proof⟩

lemma *stable_Union*:

"(!!A. A ∈ X ==> F ∈ stable A) ==> F ∈ stable (⋃ X)"

⟨proof⟩

1.0.10 Intersection

lemma *stable_Int*:

"[| F ∈ stable A; F ∈ stable A' |] ==> F ∈ stable (A ∩ A')"

⟨proof⟩

lemma *stable_INT*:

"(!!i. i ∈ I ==> F ∈ stable (A i)) ==> F ∈ stable (⋂ i ∈ I. A i)"

⟨proof⟩

lemma *stable_Inter*:

"(!!A. A ∈ X ==> F ∈ stable A) ==> F ∈ stable (⋂ X)"

⟨proof⟩

lemma *stable_constrains_Un*:

"[| F ∈ stable C; F ∈ A co (C ∪ A') |] ==> F ∈ (C ∪ A) co (C ∪ A')"

⟨proof⟩

lemma *stable_constrains_Int*:

"[| F ∈ stable C; F ∈ (C ∩ A) co A' |] ==> F ∈ (C ∩ A) co (C ∩ A')"

⟨proof⟩

lemmas *stable_constrains_stable* = *stable_constrains_Int*[THEN *stableI*]

1.0.11 invariant

lemma *invariantI*: "[| Init F ⊆ A; F ∈ stable A |] ==> F ∈ invariant A"

⟨proof⟩

Could also say *invariant A* ∩ *invariant B* ⊆ *invariant (A ∩ B)*

lemma *invariant_Int*:

"[| F ∈ invariant A; F ∈ invariant B |] ==> F ∈ invariant (A ∩ B)"

⟨proof⟩

1.0.12 increasing

lemma *increasingD*:

"F ∈ increasing f ==> F ∈ stable {s. z ⊆ f s}"

⟨proof⟩

lemma *increasing_constant* [*iff*]: " $F \in \text{increasing } (\%s. c)$ "
 <*proof*>

lemma *mono_increasing_o*:
 " $\text{mono } g \implies \text{increasing } f \subseteq \text{increasing } (g \circ f)$ "
 <*proof*>

lemma *strict_increasingD*:
 " $!!z::\text{nat}. F \in \text{increasing } f \implies F \in \text{stable } \{s. z < f s\}$ "
 <*proof*>

lemma *elimination*:
 " $[| \forall m \in M. F \in \{s. s x = m\} \text{ co } (B m) |]$
 $\implies F \in \{s. s x \in M\} \text{ co } (\bigcup_{m \in M} B m)$ "
 <*proof*>

As above, but for the trivial case of a one-variable state, in which the state is identified with its one variable.

lemma *elimination_sing*:
 " $(\forall m \in M. F \in \{m\} \text{ co } (B m)) \implies F \in M \text{ co } (\bigcup_{m \in M} B m)$ "
 <*proof*>

1.0.13 Theoretical Results from Section 6

lemma *constrains_strongest_rhs*:
 " $F \in A \text{ co } (\text{strongest_rhs } F A)$ "
 <*proof*>

lemma *strongest_rhs_is_strongest*:
 " $F \in A \text{ co } B \implies \text{strongest_rhs } F A \subseteq B$ "
 <*proof*>

1.0.14 Ad-hoc set-theory rules

lemma *Un_Diff_Diff* [*simp*]: " $A \cup B - (A - B) = B$ "
 <*proof*>

lemma *Int_Union_Union*: " $\bigcup B \cap A = \bigcup ((\%C. C \cap A) 'B)$ "
 <*proof*>

Needed for WF reasoning in WFair.thy

lemma *Image_less_than* [*simp*]: " $\text{less_than } \{k\} = \text{greaterThan } k$ "
 <*proof*>

lemma *Image_inverse_less_than* [*simp*]: " $\text{less_than}^{-1} \{k\} = \text{lessThan } k$ "
 <*proof*>

1.1 Partial versus Total Transitions

definition *totalize_act* :: " $('a * 'a)\text{set} \Rightarrow ('a * 'a)\text{set}$ " where

```
"totalize_act act == act ∪ Id_on (-(Domain act))"
```

```
definition totalize :: "'a program => 'a program" where
  "totalize F == mk_program (Init F,
                             totalize_act ' Acts F,
                             AllowedActs F)"
```

```
definition mk_total_program :: "('a set * ('a * 'a)set set * ('a * 'a)set set)
  => 'a program" where
  "mk_total_program args == totalize (mk_program args)"
```

```
definition all_total :: "'a program => bool" where
  "all_total F == ∀ act ∈ Acts F. Domain act = UNIV"
```

```
lemma insert_Id_image_Acts: "f Id = Id ==> insert Id (f'Acts F) = f ' Acts
F"
<proof>
```

1.1.1 Basic properties

```
lemma totalize_act_Id [simp]: "totalize_act Id = Id"
<proof>
```

```
lemma Domain_totalize_act [simp]: "Domain (totalize_act act) = UNIV"
<proof>
```

```
lemma Init_totalize [simp]: "Init (totalize F) = Init F"
<proof>
```

```
lemma Acts_totalize [simp]: "Acts (totalize F) = (totalize_act ' Acts F)"
<proof>
```

```
lemma AllowedActs_totalize [simp]: "AllowedActs (totalize F) = AllowedActs
F"
<proof>
```

```
lemma totalize_constrains_iff [simp]: "(totalize F ∈ A co B) = (F ∈ A co
B)"
<proof>
```

```
lemma totalize_stable_iff [simp]: "(totalize F ∈ stable A) = (F ∈ stable
A)"
<proof>
```

```
lemma totalize_invariant_iff [simp]:
  "(totalize F ∈ invariant A) = (F ∈ invariant A)"
<proof>
```

```
lemma all_total_totalize: "all_total (totalize F)"
<proof>
```

```
lemma Domain_iff_totalize_act: "(Domain act = UNIV) = (totalize_act act =
act)"
<proof>
```

```
lemma all_total_imp_totalize: "all_total F ==> (totalize F = F)"
<proof>
```

```
lemma all_total_iff_totalize: "all_total F = (totalize F = F)"
<proof>
```

```
lemma mk_total_program_constrains_iff [simp]:
  "(mk_total_program args ∈ A co B) = (mk_program args ∈ A co B)"
<proof>
```

1.2 Rules for Lazy Definition Expansion

They avoid expanding the full program, which is a large expression

```
lemma def_prg_Init:
  "F = mk_total_program (init,acts,allowed) ==> Init F = init"
<proof>
```

```
lemma def_prg_Acts:
  "F = mk_total_program (init,acts,allowed)
  ==> Acts F = insert Id (totalize_act ' acts)"
<proof>
```

```
lemma def_prg_AllowedActs:
  "F = mk_total_program (init,acts,allowed)
  ==> AllowedActs F = insert Id allowed"
<proof>
```

An action is expanded if a pair of states is being tested against it

```
lemma def_act_simp:
  "act = {(s,s'). P s s'} ==> ((s,s') ∈ act) = P s s'"
<proof>
```

A set is expanded only if an element is being tested against it

```
lemma def_set_simp: "A = B ==> (x ∈ A) = (x ∈ B)"
<proof>
```

1.2.1 Inspectors for type "program"

```
lemma Init_total_eq [simp]:
  "Init (mk_total_program (init,acts,allowed)) = init"
<proof>
```

```
lemma Acts_total_eq [simp]:
  "Acts(mk_total_program(init,acts,allowed)) = insert Id (totalize_act'acts)"
<proof>
```

```
lemma AllowedActs_total_eq [simp]:
  "AllowedActs (mk_total_program (init,acts,allowed)) = insert Id allowed"
<proof>
```

end

2 Fixed Point of a Program

theory *FP* **imports** *UNITY* **begin**

definition *FP_Orig* :: "'a program => 'a set" **where**
 "*FP_Orig* *F* == $\bigcup \{A. \forall B. F \in \text{stable } (A \cap B)\}$ "

definition *FP* :: "'a program => 'a set" **where**
 "*FP* *F* == $\{s. F \in \text{stable } \{s\}\}$ "

lemma *stable_FP_Orig_Int*: "*F* \in *stable* (*FP_Orig* *F* *Int* *B*)"
 \langle *proof* \rangle

lemma *FP_Orig_weakest*:
 " $(\bigwedge B. F \in \text{stable } (A \cap B)) \implies A \leq \text{FP_Orig } F$ "
 \langle *proof* \rangle

lemma *stable_FP_Int*: "*F* \in *stable* (*FP* *F* \cap *B*)"
 \langle *proof* \rangle

lemma *FP_equivalence*: "*FP* *F* = *FP_Orig* *F*"
 \langle *proof* \rangle

lemma *FP_weakest*:
 " $(\bigwedge B. F \in \text{stable } (A \text{ Int } B)) \implies A \leq \text{FP } F$ "
 \langle *proof* \rangle

lemma *Compl_FP*:
 " $\neg(\text{FP } F) = (\text{UN } \text{act}: \text{Acts } F. \neg\{s. \text{act}'\{s\} \leq \{s\}\})$ "
 \langle *proof* \rangle

lemma *Diff_FP*: "*A* - (*FP* *F*) = (*UN* *act*: *Acts* *F*. *A* - $\{s. \text{act}'\{s\} \leq \{s\}\})$ "
 \langle *proof* \rangle

lemma *totalize_FP* [*simp*]: "*FP* (*totalize* *F*) = *FP* *F*"
 \langle *proof* \rangle

end

3 Progress

theory *WFair* **imports** *UNITY* **begin**

The original version of this theory was based on weak fairness. (Thus, the entire UNITY development embodied this assumption, until February 2003.) Weak fairness states that if a command is enabled continuously, then it is eventually executed. Ernie Cohen suggested that I instead adopt unconditional fairness: every command is executed infinitely often.

In fact, Misra's paper on "Progress" seems to be ambiguous about the correct interpretation, and says that the two forms of fairness are equivalent. They differ only on their treatment of partial transitions, which under unconditional fairness behave magically. That is because if there are partial transitions then there may be no fair executions, making all leads-to properties hold vacuously.

Unconditional fairness has some great advantages. By distinguishing partial transitions from total ones that are the identity on part of their domain, it is more expressive. Also, by simplifying the definition of the transient property, it simplifies many proofs. A drawback is that some laws only hold under the assumption that all transitions are total. The best-known of these is the impossibility law for leads-to.

definition

— This definition specifies conditional fairness. The rest of the theory is generic to all forms of fairness. To get weak fairness, conjoin the inclusion below with $A \subseteq \text{Domain act}$, which specifies that the action is enabled over all of A .

```
transient :: "'a set => 'a program set" where
  "transient A == {F.  $\exists \text{act} \in \text{Acts } F. \text{act} \text{ 'A } \subseteq \text{-A}$ }"
```

definition

```
ensures :: "[ 'a set, 'a set ] => 'a program set" (infixl "ensures" 60)
where
```

```
"A ensures B == (A-B co A  $\cup$  B)  $\cap$  transient (A-B)"
```

inductive_set

```
leads :: "'a program => ('a set * 'a set) set"
```

— LEADS-TO constant for the inductive definition

```
for F :: "'a program"
```

```
where
```

```
Basis: "F  $\in$  A ensures B ==> (A,B)  $\in$  leads F"
```

```
| Trans: "[| (A,B)  $\in$  leads F; (B,C)  $\in$  leads F |] ==> (A,C)  $\in$  leads F"
```

```
| Union: " $\forall A \in S. (A,B) \in \text{leads } F \implies (\text{Union } S, B) \in \text{leads } F"$ 
```

```
definition leadsTo :: "[ 'a set, 'a set ] => 'a program set" (infixl "leadsTo" 60) where
```

— visible version of the LEADS-TO relation

```
"A leadsTo B == {F. (A,B)  $\in$  leads F}"
```

```
definition wlt :: "[ 'a program, 'a set ] => 'a set" where
```

— predicate transformer: the largest set that leads to B

```
"wlt F B ==  $\bigcup \{A. F \in A \text{ leadsTo } B\}"$ 
```

```
notation leadsTo (infixl " $\mapsto$ " 60)
```

3.1 transient

```
lemma stable_transient:
```

```
"[| F  $\in$  stable A; F  $\in$  transient A |] ==>  $\exists \text{act} \in \text{Acts } F. A \subseteq \text{-} (\text{Domain act})"$ 
```

```
<proof>
```

```
lemma stable_transient_empty:
```

```
"[| F  $\in$  stable A; F  $\in$  transient A; all_total F |] ==> A = {}"
```


<proof>

lemma transient_strengthen:

"[| F ∈ transient A; B ⊆ A |] ==> F ∈ transient B"

<proof>

lemma transientI:

"[| act ∈ Acts F; act' 'A ⊆ -A |] ==> F ∈ transient A"

<proof>

lemma transientE:

"[| F ∈ transient A;
 ⋀ act. [| act ∈ Acts F; act' 'A ⊆ -A |] ==> P |]
 ==> P"

<proof>

lemma transient_empty [simp]: "transient {} = UNIV"

<proof>

This equation recovers the notion of weak fairness. A totalized program satisfies a transient assertion just if the original program contains a suitable action that is also enabled.

lemma totalize_transient_iff:

"(totalize F ∈ transient A) = (∃ act ∈ Acts F. A ⊆ Domain act & act' 'A ⊆ -A)"

<proof>

lemma totalize_transientI:

"[| act ∈ Acts F; A ⊆ Domain act; act' 'A ⊆ -A |]
 ==> totalize F ∈ transient A"

<proof>

3.2 ensures

lemma ensuresI:

"[| F ∈ (A-B) co (A ∪ B); F ∈ transient (A-B) |] ==> F ∈ A ensures B"

<proof>

lemma ensuresD:

"F ∈ A ensures B ==> F ∈ (A-B) co (A ∪ B) & F ∈ transient (A-B)"

<proof>

lemma ensures_weaken_R:

"[| F ∈ A ensures A'; A' ≤ B' |] ==> F ∈ A ensures B'"

<proof>

The L-version (precondition strengthening) fails, but we have this

lemma stable_ensures_Int:

"[| F ∈ stable C; F ∈ A ensures B |]
 ==> F ∈ (C ∩ A) ensures (C ∩ B)"

<proof>

lemma stable_transient_ensures:

"[| $F \in \text{stable } A$; $F \in \text{transient } C$; $A \subseteq B \cup C$ |] $\implies F \in A$ ensures B "
 <proof>

lemma ensures_eq: " $(A \text{ ensures } B) = (A \text{ unless } B) \cap \text{transient } (A-B)$ "
 <proof>

3.3 leadsTo

lemma leadsTo_Basis [intro]: " $F \in A$ ensures $B \implies F \in A$ leadsTo B "
 <proof>

lemma leadsTo_Trans:
 "[| $F \in A$ leadsTo B ; $F \in B$ leadsTo C |] $\implies F \in A$ leadsTo C "
 <proof>

lemma leadsTo_Basis':
 "[| $F \in A$ co $A \cup B$; $F \in \text{transient } A$ |] $\implies F \in A$ leadsTo B "
 <proof>

lemma transient_imp_leadsTo: " $F \in \text{transient } A \implies F \in A$ leadsTo $(-A)$ "
 <proof>

Useful with cancellation, disjunction

lemma leadsTo_Un_duplicate: " $F \in A$ leadsTo $(A' \cup A')$ $\implies F \in A$ leadsTo A' "
 <proof>

lemma leadsTo_Un_duplicate2:
 " $F \in A$ leadsTo $(A' \cup C \cup C) \implies F \in A$ leadsTo $(A' \cup C)$ "
 <proof>

The Union introduction rule as we should have liked to state it

lemma leadsTo_Union:
 " $(\forall A. A \in S \implies F \in A \text{ leadsTo } B) \implies F \in (\bigcup S) \text{ leadsTo } B$ "
 <proof>

lemma leadsTo_Union_Int:
 " $(\forall A. A \in S \implies F \in (A \cap C) \text{ leadsTo } B) \implies F \in (\bigcup S \cap C) \text{ leadsTo } B$ "
 <proof>

lemma leadsTo_UN:
 " $(\forall i. i \in I \implies F \in (A \ i) \text{ leadsTo } B) \implies F \in (\bigcup i \in I. A \ i) \text{ leadsTo } B$ "
 <proof>

Binary union introduction rule

lemma leadsTo_Un:
 "[| $F \in A$ leadsTo C ; $F \in B$ leadsTo C |] $\implies F \in (A \cup B) \text{ leadsTo } C$ "
 <proof>

lemma single_leadsTo_I:
 " $(\forall x. x \in A \implies F \in \{x\} \text{ leadsTo } B) \implies F \in A \text{ leadsTo } B$ "
 <proof>

The INDUCTION rule as we should have liked to state it

```

lemma leadsTo_induct:
  "[| F ∈ za leadsTo zb;
    !!A B. F ∈ A ensures B ==> P A B;
    !!A B C. [| F ∈ A leadsTo B; P A B; F ∈ B leadsTo C; P B C |]
      ==> P A C;
    !!B S. ∀A ∈ S. F ∈ A leadsTo B & P A B ==> P (⋃S) B
  |] ==> P za zb"
⟨proof⟩

```

```

lemma subset_imp_ensures: "A ⊆ B ==> F ∈ A ensures B"
⟨proof⟩

```

```

lemmas subset_imp_leadsTo = subset_imp_ensures [THEN leadsTo_Basis]

```

```

lemmas leadsTo_refl = subset_refl [THEN subset_imp_leadsTo]

```

```

lemmas empty_leadsTo = empty_subsetI [THEN subset_imp_leadsTo, simp]

```

```

lemmas leadsTo_UNIV = subset_UNIV [THEN subset_imp_leadsTo, simp]

```

Lemma is the weak version: can't see how to do it in one step

```

lemma leadsTo_induct_pre_lemma:
  "[| F ∈ za leadsTo zb;
    P zb;
    !!A B. [| F ∈ A ensures B; P B |] ==> P A;
    !!S. ∀A ∈ S. P A ==> P (⋃S)
  |] ==> P za"

```

by induction on this formula

```

⟨proof⟩

```

```

lemma leadsTo_induct_pre:
  "[| F ∈ za leadsTo zb;
    P zb;
    !!A B. [| F ∈ A ensures B; F ∈ B leadsTo zb; P B |] ==> P A;
    !!S. ∀A ∈ S. F ∈ A leadsTo zb & P A ==> P (⋃S)
  |] ==> P za"
⟨proof⟩

```

```

lemma leadsTo_weaken_R: "[| F ∈ A leadsTo A'; A' ≤ B' |] ==> F ∈ A leadsTo B'"
⟨proof⟩

```

```

lemma leadsTo_weaken_L:
  "[| F ∈ A leadsTo A'; B ⊆ A |] ==> F ∈ B leadsTo A'"
⟨proof⟩

```

Distributes over binary unions

```

lemma leadsTo_Un_distrib:
  "F ∈ (A ∪ B) leadsTo C = (F ∈ A leadsTo C & F ∈ B leadsTo C)"

```

<proof>

lemma *leadsTo_UN_distrib*:

" $F \in (\bigcup i \in I. A\ i) \text{ leadsTo } B = (\forall i \in I. F \in (A\ i) \text{ leadsTo } B)$ "

<proof>

lemma *leadsTo_Union_distrib*:

" $F \in (\bigcup S) \text{ leadsTo } B = (\forall A \in S. F \in A \text{ leadsTo } B)$ "

<proof>

lemma *leadsTo_weaken*:

" $[| F \in A \text{ leadsTo } A'; B \subseteq A; A' \leq B' |] \implies F \in B \text{ leadsTo } B'$ "

<proof>

Set difference: maybe combine with *leadsTo_weaken_L??*

lemma *leadsTo_Diff*:

" $[| F \in (A-B) \text{ leadsTo } C; F \in B \text{ leadsTo } C |] \implies F \in A \text{ leadsTo } C$ "

<proof>

lemma *leadsTo_UN_UN*:

" $(\forall i. i \in I \implies F \in (A\ i) \text{ leadsTo } (A'\ i))$

$\implies F \in (\bigcup i \in I. A\ i) \text{ leadsTo } (\bigcup i \in I. A'\ i)$ "

<proof>

Binary union version

lemma *leadsTo_Un_Un*:

" $[| F \in A \text{ leadsTo } A'; F \in B \text{ leadsTo } B' |]$

$\implies F \in (A \cup B) \text{ leadsTo } (A' \cup B')$ "

<proof>

lemma *leadsTo_cancel2*:

" $[| F \in A \text{ leadsTo } (A' \cup B); F \in B \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (A' \cup B')$ "

<proof>

lemma *leadsTo_cancel_Diff2*:

" $[| F \in A \text{ leadsTo } (A' \cup B); F \in (B-A') \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (A' \cup B')$ "

<proof>

lemma *leadsTo_cancel1*:

" $[| F \in A \text{ leadsTo } (B \cup A'); F \in B \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (B' \cup A')$ "

<proof>

lemma *leadsTo_cancel_Diff1*:

" $[| F \in A \text{ leadsTo } (B \cup A'); F \in (B-A') \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (B' \cup A')$ "

<proof>

The impossibility law

lemma *leadsTo_empty*: " $[| F \in A \text{ leadsTo } \{\}; \text{all_total } F |] \implies A = \{\}$ "
 $\langle \text{proof} \rangle$

3.4 PSP: Progress-Safety-Progress

Special case of PSP: Misra's "stable conjunction"

lemma *psp_stable*:
" $[| F \in A \text{ leadsTo } A'; F \in \text{stable } B |]$
 $\implies F \in (A \cap B) \text{ leadsTo } (A' \cap B)$ "
 $\langle \text{proof} \rangle$

lemma *psp_stable2*:
" $[| F \in A \text{ leadsTo } A'; F \in \text{stable } B |]$ $\implies F \in (B \cap A) \text{ leadsTo } (B \cap A')$ "
 $\langle \text{proof} \rangle$

lemma *psp_ensures*:
" $[| F \in A \text{ ensures } A'; F \in B \text{ co } B' |]$
 $\implies F \in (A \cap B') \text{ ensures } ((A' \cap B) \cup (B' - B))$ "
 $\langle \text{proof} \rangle$

lemma *psp*:
" $[| F \in A \text{ leadsTo } A'; F \in B \text{ co } B' |]$
 $\implies F \in (A \cap B') \text{ leadsTo } ((A' \cap B) \cup (B' - B))$ "
 $\langle \text{proof} \rangle$

lemma *psp2*:
" $[| F \in A \text{ leadsTo } A'; F \in B \text{ co } B' |]$
 $\implies F \in (B' \cap A) \text{ leadsTo } ((B \cap A') \cup (B' - B))$ "
 $\langle \text{proof} \rangle$

lemma *psp_unless*:
" $[| F \in A \text{ leadsTo } A'; F \in B \text{ unless } B' |]$
 $\implies F \in (A \cap B) \text{ leadsTo } ((A' \cap B) \cup B')$ "
 $\langle \text{proof} \rangle$

3.5 Proving the induction rules

lemma *leadsTo_wf_induct_lemma*:
" $[| \text{wf } r;$
 $\forall m. F \in (A \cap f^{-1}\{m\}) \text{ leadsTo } ((A \cap f^{-1}(r^{-1}\{m\})) \cup B) |]$
 $\implies F \in (A \cap f^{-1}\{m\}) \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

lemma *leadsTo_wf_induct*:
" $[| \text{wf } r;$
 $\forall m. F \in (A \cap f^{-1}\{m\}) \text{ leadsTo } ((A \cap f^{-1}(r^{-1}\{m\})) \cup B) |]$
 $\implies F \in A \text{ leadsTo } B$ "

<proof>

```

lemma bounded_induct:
  "[| wf r;
    ∀m ∈ I. F ∈ (A ∩ f-‘{m}) leadsTo
      ((A ∩ f-‘(r-1 ‘‘ {m})) ∪ B) |]
  ==> F ∈ A leadsTo ((A - (f-‘I)) ∪ B)"
<proof>

```

```

lemma lessThan_induct:
  "[| !!m::nat. F ∈ (A ∩ f-‘{m}) leadsTo ((A ∩ f-‘{..<proof>

```

```

lemma lessThan_bounded_induct:
  "!!l::nat. [| ∀m ∈ greaterThan l.
    F ∈ (A ∩ f-‘{m}) leadsTo ((A ∩ f-‘(lessThan m)) ∪ B) |]
  ==> F ∈ A leadsTo ((A ∩ (f-‘(atMost l))) ∪ B)"
<proof>

```

```

lemma greaterThan_bounded_induct:
  "(!!l::nat. ∀m ∈ lessThan l.
    F ∈ (A ∩ f-‘{m}) leadsTo ((A ∩ f-‘(greaterThan m)) ∪ B))
  ==> F ∈ A leadsTo ((A ∩ (f-‘(atLeast l))) ∪ B)"
<proof>

```

3.6 wlt

Misra’s property W3

```

lemma wlt_leadsTo: "F ∈ (wlt F B) leadsTo B"
<proof>

```

```

lemma leadsTo_subset: "F ∈ A leadsTo B ==> A ⊆ wlt F B"
<proof>

```

Misra’s property W2

```

lemma leadsTo_eq_subset_wlt: "F ∈ A leadsTo B = (A ⊆ wlt F B)"
<proof>

```

Misra’s property W4

```

lemma wlt_increasing: "B ⊆ wlt F B"
<proof>

```

Used in the Trans case below

```

lemma lemma1:
  "[| B ⊆ A2;
    F ∈ (A1 - B) co (A1 ∪ B);
    F ∈ (A2 - C) co (A2 ∪ C) |]
  ==> F ∈ (A1 ∪ A2 - C) co (A1 ∪ A2 ∪ C)"

```

<proof>

Lemma (1,2,3) of Misra's draft book, Chapter 4, "Progress"

lemma *leadsTo_123*:

" $F \in A$ leadsTo A'
 $\implies \exists B. A \subseteq B \ \& \ F \in B$ leadsTo A' & $F \in (B-A')$ co $(B \cup A')$ "

<proof>

Misra's property W5

lemma *wlt_constrains_wlt*: " $F \in (wlt \ F \ B - B)$ co $(wlt \ F \ B)$ "

<proof>

3.7 Completion: Binary and General Finite versions

lemma *completion_lemma* :

" $[| \ W = wlt \ F \ (B' \cup C);$
 $F \in A$ leadsTo $(A' \cup C); \ F \in A'$ co $(A' \cup C);$
 $F \in B$ leadsTo $(B' \cup C); \ F \in B'$ co $(B' \cup C) \ |]$
 $\implies F \in (A \cap B)$ leadsTo $((A' \cap B') \cup C)$ "

<proof>

lemmas *completion = completion_lemma* [*OF refl*]

lemma *finite_completion_lemma*:

"*finite* $I \implies (\forall i \in I. F \in (A \ i)$ leadsTo $(A' \ i \cup C)$ \implies
 $(\forall i \in I. F \in (A' \ i)$ co $(A' \ i \cup C)$) \implies
 $F \in (\bigcap i \in I. A \ i)$ leadsTo $((\bigcap i \in I. A' \ i) \cup C)$ "

<proof>

lemma *finite_completion*:

" $[|$ *finite* $I;$
 $!!i. i \in I \implies F \in (A \ i)$ leadsTo $(A' \ i \cup C);$
 $!!i. i \in I \implies F \in (A' \ i)$ co $(A' \ i \cup C) \ |]$
 $\implies F \in (\bigcap i \in I. A \ i)$ leadsTo $((\bigcap i \in I. A' \ i) \cup C)$ "

<proof>

lemma *stable_completion*:

" $[| F \in A$ leadsTo $A'; \ F \in$ stable $A';$
 $F \in B$ leadsTo $B'; \ F \in$ stable $B' \ |]$
 $\implies F \in (A \cap B)$ leadsTo $(A' \cap B')$ "

<proof>

lemma *finite_stable_completion*:

" $[|$ *finite* $I;$
 $!!i. i \in I \implies F \in (A \ i)$ leadsTo $(A' \ i);$
 $!!i. i \in I \implies F \in$ stable $(A' \ i) \ |]$
 $\implies F \in (\bigcap i \in I. A \ i)$ leadsTo $(\bigcap i \in I. A' \ i)$ "

<proof>

end

4 Weak Safety

theory *Constrains* imports *UNITY* begin

inductive_set

```
traces :: "[ 'a set, ('a * 'a) set set ] => ('a * 'a list) set"
for init :: "'a set" and acts :: "('a * 'a) set set"
where
```

```
Init: "s ∈ init ==> (s, []) ∈ traces init acts"
```

```
| Acts: "[| act ∈ acts; (s, evs) ∈ traces init acts; (s, s') ∈ act |]
==> (s', s#evs) ∈ traces init acts"
```

inductive_set

```
reachable :: "'a program => 'a set"
for F :: "'a program"
where
```

```
Init: "s ∈ Init F ==> s ∈ reachable F"
```

```
| Acts: "[| act ∈ Acts F; s ∈ reachable F; (s, s') ∈ act |]
==> s' ∈ reachable F"
```

definition *Constrains* :: "['a set, 'a set] => 'a program set" (infixl "Co" 60)
where

```
"A Co B == {F. F ∈ (reachable F ∩ A) co B}"
```

definition *Unless* :: "['a set, 'a set] => 'a program set" (infixl "Unless" 60) where

```
"A Unless B == (A-B) Co (A ∪ B)"
```

definition *Stable* :: "'a set => 'a program set" where

```
"Stable A == A Co A"
```

definition *Always* :: "'a set => 'a program set" where

```
"Always A == {F. Init F ⊆ A} ∩ Stable A"
```

definition *Increasing* :: "['a => 'b::order] => 'a program set" where

```
"Increasing f == ⋂ z. Stable {s. z ≤ f s}"
```

4.1 traces and reachable

lemma *reachable_equiv_traces*:

```
"reachable F = {s. ∃ evs. (s, evs) ∈ traces (Init F) (Acts F)}"
⟨proof⟩
```

lemma *Init_subset_reachable*: "Init F ⊆ reachable F"

⟨proof⟩

lemma *stable_reachable* [intro!,simp]:
 "Acts $G \subseteq$ Acts $F \implies G \in$ stable (reachable F)"
 <proof>

lemma *invariant_reachable*: " $F \in$ invariant (reachable F)"
 <proof>

lemma *invariant_includes_reachable*: " $F \in$ invariant $A \implies$ reachable $F \subseteq A$ "
 <proof>

4.2 Co

lemmas *constrains_reachable_Int* =
 subset_refl [THEN *stable_reachable* [unfolded *stable_def*], THEN *constrains_Int*]

lemma *Constrains_eq_constrains*:
 " $A \text{ Co } B = \{F. F \in (\text{reachable } F \cap A) \text{ co } (\text{reachable } F \cap B)\}$ "
 <proof>

lemma *constrains_imp_Constrains*: " $F \in A \text{ co } A' \implies F \in A \text{ Co } A'$ "
 <proof>

lemma *stable_imp_Stable*: " $F \in$ stable $A \implies F \in$ Stable A "
 <proof>

lemma *ConstrainsI*:
 "(!!act $s \ s'. [| \text{act} \in \text{Acts } F; (s,s') \in \text{act}; s \in A |] \implies s' \in A')$
 $\implies F \in A \text{ Co } A'$ "
 <proof>

lemma *Constrains_empty* [iff]: " $F \in \{\} \text{ Co } B$ "
 <proof>

lemma *Constrains_UNIV* [iff]: " $F \in A \text{ Co UNIV}$ "
 <proof>

lemma *Constrains_weaken_R*:
 " $[| F \in A \text{ Co } A'; A' \leq B' |] \implies F \in A \text{ Co } B'$ "
 <proof>

lemma *Constrains_weaken_L*:
 " $[| F \in A \text{ Co } A'; B \subseteq A |] \implies F \in B \text{ Co } A'$ "
 <proof>

lemma *Constrains_weaken*:
 " $[| F \in A \text{ Co } A'; B \subseteq A; A' \leq B' |] \implies F \in B \text{ Co } B'$ "
 <proof>

lemma *Constrains_Un*:

"[| $F \in A \text{ Co } A'$; $F \in B \text{ Co } B'$ |] $\implies F \in (A \cup B) \text{ Co } (A' \cup B')$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_UN*:

assumes *Co*: " $\forall i. i \in I \implies F \in (A \ i) \text{ Co } (A' \ i)$ "
shows " $F \in (\bigcup i \in I. A \ i) \text{ Co } (\bigcup i \in I. A' \ i)$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_Int*:

"[| $F \in A \text{ Co } A'$; $F \in B \text{ Co } B'$ |] $\implies F \in (A \cap B) \text{ Co } (A' \cap B')$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_INT*:

assumes *Co*: " $\forall i. i \in I \implies F \in (A \ i) \text{ Co } (A' \ i)$ "
shows " $F \in (\bigcap i \in I. A \ i) \text{ Co } (\bigcap i \in I. A' \ i)$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_imp_subset*: " $F \in A \text{ Co } A' \implies \text{reachable } F \cap A \subseteq A'$ "

$\langle \text{proof} \rangle$

lemma *Constrains_trans*: "[| $F \in A \text{ Co } B$; $F \in B \text{ Co } C$ |] $\implies F \in A \text{ Co } C$ "

$\langle \text{proof} \rangle$

lemma *Constrains_cancel*:

"[| $F \in A \text{ Co } (A' \cup B)$; $F \in B \text{ Co } B'$ |] $\implies F \in A \text{ Co } (A' \cup B')$ "
 $\langle \text{proof} \rangle$

4.3 Stable

lemma *Stable_eq*: "[| $F \in \text{Stable } A$; $A = B$ |] $\implies F \in \text{Stable } B$ "

$\langle \text{proof} \rangle$

lemma *Stable_eq_stable*: " $(F \in \text{Stable } A) = (F \in \text{stable } (\text{reachable } F \cap A))$ "

$\langle \text{proof} \rangle$

lemma *StableI*: " $F \in A \text{ Co } A \implies F \in \text{Stable } A$ "

$\langle \text{proof} \rangle$

lemma *StableD*: " $F \in \text{Stable } A \implies F \in A \text{ Co } A$ "

$\langle \text{proof} \rangle$

lemma *Stable_Un*:

"[| $F \in \text{Stable } A$; $F \in \text{Stable } A'$ |] $\implies F \in \text{Stable } (A \cup A')$ "
 $\langle \text{proof} \rangle$

lemma *Stable_Int*:

"[| $F \in \text{Stable } A$; $F \in \text{Stable } A'$ |] $\implies F \in \text{Stable } (A \cap A')$ "
 $\langle \text{proof} \rangle$

lemma *Stable_Constrains_Un*:

"[| $F \in \text{Stable } C$; $F \in A \text{ Co } (C \cup A')$ |]

$\implies F \in (C \cup A) \text{ Co } (C \cup A')$
 $\langle \text{proof} \rangle$

lemma *Stable_Constrains_Int*:
 $"[| F \in \text{Stable } C; F \in (C \cap A) \text{ Co } A' |]$
 $\implies F \in (C \cap A) \text{ Co } (C \cap A)'"$
 $\langle \text{proof} \rangle$

lemma *Stable_UN*:
 $"(!i. i \in I \implies F \in \text{Stable } (A \ i)) \implies F \in \text{Stable } (\bigcup i \in I. A \ i)"$
 $\langle \text{proof} \rangle$

lemma *Stable_INT*:
 $"(!i. i \in I \implies F \in \text{Stable } (A \ i)) \implies F \in \text{Stable } (\bigcap i \in I. A \ i)"$
 $\langle \text{proof} \rangle$

lemma *Stable_reachable*: $"F \in \text{Stable } (\text{reachable } F)"$
 $\langle \text{proof} \rangle$

4.4 Increasing

lemma *IncreasingD*:
 $"F \in \text{Increasing } f \implies F \in \text{Stable } \{s. x \leq f \ s\}"$
 $\langle \text{proof} \rangle$

lemma *mono_Increasing_o*:
 $"\text{mono } g \implies \text{Increasing } f \subseteq \text{Increasing } (g \circ f)"$
 $\langle \text{proof} \rangle$

lemma *strict_IncreasingD*:
 $"!!z::nat. F \in \text{Increasing } f \implies F \in \text{Stable } \{s. z < f \ s\}"$
 $\langle \text{proof} \rangle$

lemma *increasing_imp_Increasing*:
 $"F \in \text{increasing } f \implies F \in \text{Increasing } f"$
 $\langle \text{proof} \rangle$

lemmas *Increasing_constant = increasing_constant [THEN increasing_imp_Increasing, iff]*

4.5 The Elimination Theorem

lemma *Elimination*:
 $"[| \forall m. F \in \{s. s \ x = m\} \text{ Co } (B \ m) |]$
 $\implies F \in \{s. s \ x \in M\} \text{ Co } (\bigcup m \in M. B \ m)"$
 $\langle \text{proof} \rangle$

lemma *Elimination_sing*:
 $"(\forall m. F \in \{m\} \text{ Co } (B \ m)) \implies F \in M \text{ Co } (\bigcup m \in M. B \ m)"$
 $\langle \text{proof} \rangle$

4.6 Specialized laws for handling Always

lemma *AlwaysI*: "[| Init F \subseteq A; F \in Stable A |] ==> F \in Always A"
 <proof>

lemma *AlwaysD*: "F \in Always A ==> Init F \subseteq A & F \in Stable A"
 <proof>

lemmas *AlwaysE* = *AlwaysD* [THEN conjE]
lemmas *Always_imp_Stable* = *AlwaysD* [THEN conjunct2]

lemma *Always_includes_reachable*: "F \in Always A ==> reachable F \subseteq A"
 <proof>

lemma *invariant_imp_Always*:
 "F \in invariant A ==> F \in Always A"
 <proof>

lemmas *Always_reachable* = *invariant_reachable* [THEN *invariant_imp_Always*]

lemma *Always_eq_invariant_reachable*:
 "Always A = {F. F \in invariant (reachable F \cap A)}"
 <proof>

lemma *Always_eq_includes_reachable*: "Always A = {F. reachable F \subseteq A}"
 <proof>

lemma *Always_UNIV_eq* [simp]: "Always UNIV = UNIV"
 <proof>

lemma *UNIV_AlwaysI*: "UNIV \subseteq A ==> F \in Always A"
 <proof>

lemma *Always_eq_UN_invariant*: "Always A = (\bigcup I \in Pow A. invariant I)"
 <proof>

lemma *Always_weaken*: "[| F \in Always A; A \subseteq B |] ==> F \in Always B"
 <proof>

4.7 "Co" rules involving Always

lemma *Always_Constrains_pre*:
 "F \in Always INV ==> (F \in (INV \cap A) Co A') = (F \in A Co A')"
 <proof>

lemma *Always_Constrains_post*:
 "F \in Always INV ==> (F \in A Co (INV \cap A')) = (F \in A Co A')"
 <proof>

lemmas *Always_ConstrainsI* = *Always_Constrains_pre* [THEN iffD1]

lemmas Always_ConstrainsD = Always_Constrains_post [THEN iffD2]

lemma Always_Constrains_weaken:
 "[| F ∈ Always C; F ∈ A Co A';
 C ∩ B ⊆ A; C ∩ A' ⊆ B' |]
 ==> F ∈ B Co B'"
 <proof>

lemma Always_Int_distrib: "Always (A ∩ B) = Always A ∩ Always B"
 <proof>

lemma Always_INT_distrib: "Always (∩ (A ' I)) = (∩ i ∈ I. Always (A i))"
 <proof>

lemma Always_Int_I:
 "[| F ∈ Always A; F ∈ Always B |] ==> F ∈ Always (A ∩ B)"
 <proof>

lemma Always_Compl_Un_eq:
 "F ∈ Always A ==> (F ∈ Always (-A ∪ B)) = (F ∈ Always B)"
 <proof>

lemmas Always_thin = thin_rl [of "F ∈ Always A"] for F A

4.8 Totalize

lemma reachable_imp_reachable_tot:
 "s ∈ reachable F ==> s ∈ reachable (totalize F)"
 <proof>

lemma reachable_tot_imp_reachable:
 "s ∈ reachable (totalize F) ==> s ∈ reachable F"
 <proof>

lemma reachable_tot_eq [simp]: "reachable (totalize F) = reachable F"
 <proof>

lemma totalize_Constrains_iff [simp]: "(totalize F ∈ A Co B) = (F ∈ A Co B)"
 <proof>

lemma totalize_Stable_iff [simp]: "(totalize F ∈ Stable A) = (F ∈ Stable A)"
 <proof>

lemma totalize_Always_iff [simp]: "(totalize F ∈ Always A) = (F ∈ Always A)"
 <proof>

<proof>

end

5 Weak Progress

theory SubstAx imports WFair Constrains begin

definition Ensures :: "['a set, 'a set] => 'a program set" (infixl "Ensures" 60) where
 "A Ensures B == {F. F ∈ (reachable F ∩ A) ensures B}"

definition LeadsTo :: "['a set, 'a set] => 'a program set" (infixl "LeadsTo" 60) where
 "A LeadsTo B == {F. F ∈ (reachable F ∩ A) leadsTo B}"

notation LeadsTo (infixl "⟶_w" 60)

Resembles the previous definition of LeadsTo

lemma LeadsTo_eq_leadsTo:
 "A LeadsTo B = {F. F ∈ (reachable F ∩ A) leadsTo (reachable F ∩ B)}"
<proof>

5.1 Specialized laws for handling invariants

lemma Always_LeadsTo_pre:
 "F ∈ Always INV ==> (F ∈ (INV ∩ A) LeadsTo A') = (F ∈ A LeadsTo A')"
<proof>

lemma Always_LeadsTo_post:
 "F ∈ Always INV ==> (F ∈ A LeadsTo (INV ∩ A')) = (F ∈ A LeadsTo A')"
<proof>

lemmas Always_LeadsToI = Always_LeadsTo_pre [THEN iffD1]

lemmas Always_LeadsToD = Always_LeadsTo_post [THEN iffD2]

5.2 Introduction rules: Basis, Trans, Union

lemma leadsTo_imp_LeadsTo: "F ∈ A leadsTo B ==> F ∈ A LeadsTo B"
<proof>

lemma LeadsTo_Trans:
 "[| F ∈ A LeadsTo B; F ∈ B LeadsTo C |] ==> F ∈ A LeadsTo C"
<proof>

lemma LeadsTo_Union:
 "(!A. A ∈ S ==> F ∈ A LeadsTo B) ==> F ∈ (⋃ S) LeadsTo B"
<proof>

5.3 Derived rules

lemma *LeadsTo_UNIV [simp]: "F ∈ A LeadsTo UNIV"*
 ⟨proof⟩

Useful with cancellation, disjunction

lemma *LeadsTo_Un_duplicate:*
 "F ∈ A LeadsTo (A' ∪ A') ==> F ∈ A LeadsTo A'"
 ⟨proof⟩

lemma *LeadsTo_Un_duplicate2:*
 "F ∈ A LeadsTo (A' ∪ C ∪ C) ==> F ∈ A LeadsTo (A' ∪ C)"
 ⟨proof⟩

lemma *LeadsTo_UN:*
 "(!!i. i ∈ I ==> F ∈ (A i) LeadsTo B) ==> F ∈ (⋃ i ∈ I. A i) LeadsTo B"
 ⟨proof⟩

Binary union introduction rule

lemma *LeadsTo_Un:*
 "[| F ∈ A LeadsTo C; F ∈ B LeadsTo C |] ==> F ∈ (A ∪ B) LeadsTo C"
 ⟨proof⟩

Lets us look at the starting state

lemma *single_LeadsTo_I:*
 "(!!s. s ∈ A ==> F ∈ {s} LeadsTo B) ==> F ∈ A LeadsTo B"
 ⟨proof⟩

lemma *subset_imp_LeadsTo: "A ⊆ B ==> F ∈ A LeadsTo B"*
 ⟨proof⟩

lemmas *empty_LeadsTo = empty_subsetI [THEN subset_imp_LeadsTo, simp]*

lemma *LeadsTo_weaken_R:*
 "[| F ∈ A LeadsTo A'; A' ⊆ B' |] ==> F ∈ A LeadsTo B'"
 ⟨proof⟩

lemma *LeadsTo_weaken_L:*
 "[| F ∈ A LeadsTo A'; B ⊆ A |]
 ==> F ∈ B LeadsTo A'"
 ⟨proof⟩

lemma *LeadsTo_weaken:*
 "[| F ∈ A LeadsTo A';
 B ⊆ A; A' ⊆ B' |]
 ==> F ∈ B LeadsTo B'"
 ⟨proof⟩

lemma *Always_LeadsTo_weaken:*
 "[| F ∈ Always C; F ∈ A LeadsTo A';
 C ∩ B ⊆ A; C ∩ A' ⊆ B' |]
 ==> F ∈ B LeadsTo B'"
 ⟨proof⟩

lemma LeadsTo_Un_post: " $F \in A$ LeadsTo $B \implies F \in (A \cup B)$ LeadsTo B "
 <proof>

lemma LeadsTo_Trans_Un:
 "[| $F \in A$ LeadsTo B ; $F \in B$ LeadsTo C |]
 $\implies F \in (A \cup B)$ LeadsTo C "
 <proof>

lemma LeadsTo_Un_distrib:
 " $(F \in (A \cup B)$ LeadsTo $C) = (F \in A$ LeadsTo $C \ \& \ F \in B$ LeadsTo $C)$ "
 <proof>

lemma LeadsTo_UN_distrib:
 " $(F \in (\bigcup_{i \in I} A \ i)$ LeadsTo $B) = (\forall i \in I. F \in (A \ i)$ LeadsTo $B)$ "
 <proof>

lemma LeadsTo_Union_distrib:
 " $(F \in (\bigcup S)$ LeadsTo $B) = (\forall A \in S. F \in A$ LeadsTo $B)$ "
 <proof>

lemma LeadsTo_Basis: " $F \in A$ Ensures $B \implies F \in A$ LeadsTo B "
 <proof>

lemma EnsuresI:
 "[| $F \in (A-B)$ Co $(A \cup B)$; $F \in$ transient $(A-B)$ |]
 $\implies F \in A$ Ensures B "
 <proof>

lemma Always_LeadsTo_Basis:
 "[| $F \in$ Always INV ;
 $F \in (INV \cap (A-A'))$ Co $(A \cup A')$;
 $F \in$ transient $(INV \cap (A-A'))$ |]
 $\implies F \in A$ LeadsTo A' "
 <proof>

Set difference: maybe combine with `leadsTo_weaken_L`?? This is the most useful form of the "disjunction" rule

lemma LeadsTo_Diff:
 "[| $F \in (A-B)$ LeadsTo C ; $F \in (A \cap B)$ LeadsTo C |]
 $\implies F \in A$ LeadsTo C "
 <proof>

lemma LeadsTo_UN_UN:
 " $(!! i. i \in I \implies F \in (A \ i)$ LeadsTo $(A' \ i))$ "

$\implies F \in (\bigcup i \in I. A \ i) \text{ LeadsTo } (\bigcup i \in I. A' \ i)$
 <proof>

Version with no index set

lemma *LeadsTo_UN_UN_noindex*:
 " $(\forall i. F \in (A \ i) \text{ LeadsTo } (A' \ i)) \implies F \in (\bigcup i. A \ i) \text{ LeadsTo } (\bigcup i. A' \ i)$ "
 <proof>

Version with no index set

lemma *all_LeadsTo_UN_UN*:
 " $\forall i. F \in (A \ i) \text{ LeadsTo } (A' \ i) \implies F \in (\bigcup i. A \ i) \text{ LeadsTo } (\bigcup i. A' \ i)$ "
 <proof>

Binary union version

lemma *LeadsTo_Un_Un*:
 " $[| F \in A \text{ LeadsTo } A'; F \in B \text{ LeadsTo } B' \ |] \implies F \in (A \cup B) \text{ LeadsTo } (A' \cup B')$ "
 <proof>

lemma *LeadsTo_cancel2*:
 " $[| F \in A \text{ LeadsTo } (A' \cup B); F \in B \text{ LeadsTo } B' \ |] \implies F \in A \text{ LeadsTo } (A' \cup B')$ "
 <proof>

lemma *LeadsTo_cancel_Diff2*:
 " $[| F \in A \text{ LeadsTo } (A' \cup B); F \in (B-A') \text{ LeadsTo } B' \ |] \implies F \in A \text{ LeadsTo } (A' \cup B')$ "
 <proof>

lemma *LeadsTo_cancel1*:
 " $[| F \in A \text{ LeadsTo } (B \cup A'); F \in B \text{ LeadsTo } B' \ |] \implies F \in A \text{ LeadsTo } (B' \cup A')$ "
 <proof>

lemma *LeadsTo_cancel_Diff1*:
 " $[| F \in A \text{ LeadsTo } (B \cup A'); F \in (B-A') \text{ LeadsTo } B' \ |] \implies F \in A \text{ LeadsTo } (B' \cup A')$ "
 <proof>

The impossibility law

The set "A" may be non-empty, but it contains no reachable states

lemma *LeadsTo_empty*: " $[| F \in A \text{ LeadsTo } \{\}; \text{all_total } F \ |] \implies F \in \text{Always } (-A)$ "
 <proof>

5.4 PSP: Progress-Safety-Progress

Special case of PSP: Misra's "stable conjunction"

```

lemma PSP_Stable:
  "[| F ∈ A LeadsTo A'; F ∈ Stable B |]
  ==> F ∈ (A ∩ B) LeadsTo (A' ∩ B)"
  <proof>

lemma PSP_Stable2:
  "[| F ∈ A LeadsTo A'; F ∈ Stable B |]
  ==> F ∈ (B ∩ A) LeadsTo (B ∩ A')"
  <proof>

lemma PSP:
  "[| F ∈ A LeadsTo A'; F ∈ B Co B' |]
  ==> F ∈ (A ∩ B') LeadsTo ((A' ∩ B) ∪ (B' - B))"
  <proof>

lemma PSP2:
  "[| F ∈ A LeadsTo A'; F ∈ B Co B' |]
  ==> F ∈ (B' ∩ A) LeadsTo ((B ∩ A') ∪ (B' - B))"
  <proof>

lemma PSP_Unless:
  "[| F ∈ A LeadsTo A'; F ∈ B Unless B' |]
  ==> F ∈ (A ∩ B) LeadsTo ((A' ∩ B) ∪ B')"
  <proof>

lemma Stable_transient_Always_LeadsTo:
  "[| F ∈ Stable A; F ∈ transient C;
  F ∈ Always (-A ∪ B ∪ C) |] ==> F ∈ A LeadsTo B"
  <proof>

```

5.5 Induction rules

```

lemma LeadsTo_wf_induct:
  "[| wf r;
  ∀m. F ∈ (A ∩ f-'{m}) LeadsTo
  ((A ∩ f-'(r-1 '' {m})) ∪ B) |]
  ==> F ∈ A LeadsTo B"
  <proof>

lemma Bounded_induct:
  "[| wf r;
  ∀m ∈ I. F ∈ (A ∩ f-'{m}) LeadsTo
  ((A ∩ f-'(r-1 '' {m})) ∪ B) |]
  ==> F ∈ A LeadsTo ((A - (f-I)) ∪ B)"
  <proof>

lemma LessThan_induct:
  "(!!m::nat. F ∈ (A ∩ f-'{m}) LeadsTo ((A ∩ f-'(lessThan m)) ∪ B))
  ==> F ∈ A LeadsTo B"
  <proof>

```

Integer version. Could generalize from 0 to any lower bound

lemma *integ_0_le_induct*:

```
"[| F ∈ Always {s. (0::int) ≤ f s};
  !! z. F ∈ (A ∩ {s. f s = z}) LeadsTo
            ((A ∩ {s. f s < z}) ∪ B) |]
  ==> F ∈ A LeadsTo B"
```

<proof>

lemma *LessThan_bounded_induct*:

```
"!!l::nat. ∀m ∈ greaterThan l.
  F ∈ (A ∩ f-'{m}) LeadsTo ((A ∩ f-'(lessThan m)) ∪ B)
  ==> F ∈ A LeadsTo ((A ∩ (f-'(atMost l))) ∪ B)"
```

<proof>

lemma *GreaterThan_bounded_induct*:

```
"!!l::nat. ∀m ∈ lessThan l.
  F ∈ (A ∩ f-'{m}) LeadsTo ((A ∩ f-'(greaterThan m)) ∪ B)
  ==> F ∈ A LeadsTo ((A ∩ (f-'(atLeast l))) ∪ B)"
```

<proof>

5.6 Completion: Binary and General Finite versions

lemma *Completion*:

```
"[| F ∈ A LeadsTo (A' ∪ C); F ∈ A' Co (A' ∪ C);
  F ∈ B LeadsTo (B' ∪ C); F ∈ B' Co (B' ∪ C) |]
  ==> F ∈ (A ∩ B) LeadsTo ((A' ∩ B') ∪ C)"
```

<proof>

lemma *Finite_completion_lemma*:

```
"finite I
  ==> (∀i ∈ I. F ∈ (A i) LeadsTo (A' i ∪ C)) -->
      (∀i ∈ I. F ∈ (A' i) Co (A' i ∪ C)) -->
      F ∈ (∩i ∈ I. A i) LeadsTo ((∩i ∈ I. A' i) ∪ C)"
```

<proof>

lemma *Finite_completion*:

```
"[| finite I;
  !!i. i ∈ I ==> F ∈ (A i) LeadsTo (A' i ∪ C);
  !!i. i ∈ I ==> F ∈ (A' i) Co (A' i ∪ C) |]
  ==> F ∈ (∩i ∈ I. A i) LeadsTo ((∩i ∈ I. A' i) ∪ C)"
```

<proof>

lemma *Stable_completion*:

```
"[| F ∈ A LeadsTo A'; F ∈ Stable A';
  F ∈ B LeadsTo B'; F ∈ Stable B' |]
  ==> F ∈ (A ∩ B) LeadsTo (A' ∩ B'"
```

<proof>

lemma *Finite_stable_completion*:

```
"[| finite I;
  !!i. i ∈ I ==> F ∈ (A i) LeadsTo (A' i);
  !!i. i ∈ I ==> F ∈ Stable (A' i) |]
  ==> F ∈ (∩i ∈ I. A i) LeadsTo (∩i ∈ I. A' i)"
```

<proof>

end

6 The Detects Relation

theory *Detects* imports *FP SubstAx* begin

definition *Detects* :: "[*'a set, 'a set*] => *'a program set*" (infixl "*Detects*" 60)

where "*A Detects B* = (*Always* ($\neg A \cup B$)) \cap (*B LeadsTo A*)"

definition *Equality* :: "[*'a set, 'a set*] => *'a set*" (infixl " $\langle \Leftarrow \Rightarrow \rangle$ " 60)

where "*A $\langle \Leftarrow \Rightarrow \rangle$ B* = ($\neg A \cup B$) \cap ($A \cup \neg B$)"

lemma *Always_at_FP*:

"[*F* \in *A LeadsTo B*; *all_total F*] ==> *F* \in *Always* (\neg (*FP F*) \cap *A* \cap $\neg B$)"

<proof>

lemma *Detects_Trans*:

"[*F* \in *A Detects B*; *F* \in *B Detects C*] ==> *F* \in *A Detects C*"

<proof>

lemma *Detects_refl*: "*F* \in *A Detects A*"

<proof>

lemma *Detects_eq_Un*: "*(A $\langle \Leftarrow \Rightarrow \rangle$ B)* = (*A* \cap *B*) \cup ($\neg A$ \cap $\neg B$)"

<proof>

lemma *Detects_antisym*:

"[*F* \in *A Detects B*; *F* \in *B Detects A*] ==> *F* \in *Always* (*A $\langle \Leftarrow \Rightarrow \rangle$ B*)"

<proof>

lemma *Detects_Always*:

"[*F* \in *A Detects B*; *all_total F*] ==> *F* \in *Always* (\neg (*FP F*) \cup (*A $\langle \Leftarrow \Rightarrow \rangle$ B*))"

<proof>

lemma *Detects_Imp_LeadstoEQ*:

"*F* \in *A Detects B* ==> *F* \in *UNIV LeadsTo* (*A $\langle \Leftarrow \Rightarrow \rangle$ B*)"

<proof>

end

7 Unions of Programs

theory *Union* imports *SubstAx FP* begin

definition

```
ok :: "[ 'a program, 'a program ] => bool"      (infixl "ok" 65)
where "F ok G == Acts F ⊆ AllowedActs G &
      Acts G ⊆ AllowedActs F"
```

definition

```
OK :: "[ 'a set, 'a => 'b program ] => bool"
where "OK I F = (∀ i ∈ I. ∀ j ∈ I - {i}. Acts (F i) ⊆ AllowedActs (F j))"
```

definition

```
JOIN :: "[ 'a set, 'a => 'b program ] => 'b program"
where "JOIN I F = mk_program (∩ i ∈ I. Init (F i), ∪ i ∈ I. Acts (F i),
                              ∩ i ∈ I. AllowedActs (F i))"
```

definition

```
Join :: "[ 'a program, 'a program ] => 'a program"      (infixl "⊔" 65)
where "F ⊔ G = mk_program (Init F ∩ Init G, Acts F ∪ Acts G,
                          AllowedActs F ∩ AllowedActs G)"
```

definition *SKIP* :: "'a program" ("⊥")

```
where "⊥ = mk_program (UNIV, {}, UNIV)"
```

definition

```
safety_prop :: "'a program set => bool"
where "safety_prop X ↔ SKIP ∈ X ∧ (∀ G. Acts G ⊆ ∪ (Acts ' X) → G ∈ X)"
```

syntax

```
"_JOIN1" :: "[pttrns, 'b set] => 'b set"      ("(3⊔_./_)" 10)
"_JOIN"  :: "[pttrn, 'a set, 'b set] => 'b set" ("(3⊔_∈_./_)" 10)
```

translations

```
"⊔ x ∈ A. B" == "CONST JOIN A (λx. B)"
"⊔ x y. B" == "⊔ x. ⊔ y. B"
"⊔ x. B" == "CONST JOIN (CONST UNIV) (λx. B)"
```

7.1 SKIP

lemma *Init_SKIP* [simp]: "Init SKIP = UNIV"
 <proof>

lemma *Acts_SKIP* [simp]: "Acts SKIP = {Id}"
 <proof>

lemma *AllowedActs_SKIP* [simp]: "AllowedActs SKIP = UNIV"
 <proof>

lemma *reachable_SKIP* [simp]: "reachable SKIP = UNIV"

<proof>

7.2 SKIP and safety properties

lemma *SKIP_in_constrains_iff [iff]: "(SKIP ∈ A co B) = (A ⊆ B)"*
<proof>

lemma *SKIP_in_Constrains_iff [iff]: "(SKIP ∈ A Co B) = (A ⊆ B)"*
<proof>

lemma *SKIP_in_stable [iff]: "SKIP ∈ stable A"*
<proof>

declare *SKIP_in_stable [THEN stable_imp_Stable, iff]*

7.3 Join

lemma *Init_Join [simp]: "Init (F ⊔ G) = Init F ∩ Init G"*
<proof>

lemma *Acts_Join [simp]: "Acts (F ⊔ G) = Acts F ∪ Acts G"*
<proof>

lemma *AllowedActs_Join [simp]:*
"AllowedActs (F ⊔ G) = AllowedActs F ∩ AllowedActs G"
<proof>

7.4 JN

lemma *JN_empty [simp]: "(⊔ i ∈ {}. F i) = SKIP"*
<proof>

lemma *JN_insert [simp]: "(⊔ i ∈ insert a I. F i) = (F a) ⊔ (⊔ i ∈ I. F i)"*
<proof>

lemma *Init_JN [simp]: "Init (⊔ i ∈ I. F i) = (⋂ i ∈ I. Init (F i))"*
<proof>

lemma *Acts_JN [simp]: "Acts (⊔ i ∈ I. F i) = insert Id (⋃ i ∈ I. Acts (F i))"*
<proof>

lemma *AllowedActs_JN [simp]:*
"AllowedActs (⊔ i ∈ I. F i) = (⋂ i ∈ I. AllowedActs (F i))"
<proof>

lemma *JN_cong [cong]:*
"[| I=J; !!i. i ∈ J ==> F i = G i |] ==> (⊔ i ∈ I. F i) = (⊔ i ∈ J. G i)"
<proof>

7.5 Algebraic laws

lemma *Join_commute*: " $F \sqcup G = G \sqcup F$ "

<proof>

lemma *Join_assoc*: " $(F \sqcup G) \sqcup H = F \sqcup (G \sqcup H)$ "

<proof>

lemma *Join_left_commute*: " $A \sqcup (B \sqcup C) = B \sqcup (A \sqcup C)$ "

<proof>

lemma *Join_SKIP_left [simp]*: " $SKIP \sqcup F = F$ "

<proof>

lemma *Join_SKIP_right [simp]*: " $F \sqcup SKIP = F$ "

<proof>

lemma *Join_absorb [simp]*: " $F \sqcup F = F$ "

<proof>

lemma *Join_left_absorb*: " $F \sqcup (F \sqcup G) = F \sqcup G$ "

<proof>

lemmas *Join_ac* = *Join_assoc Join_left_absorb Join_commute Join_left_commute*

7.6 Laws Governing \sqcup

lemma *JN_absorb*: " $k \in I \implies F \ k \sqcup (\sqcup i \in I. F \ i) = (\sqcup i \in I. F \ i)$ "

<proof>

lemma *JN_Un*: " $(\sqcup i \in I \cup J. F \ i) = ((\sqcup i \in I. F \ i) \sqcup (\sqcup i \in J. F \ i))$ "

<proof>

lemma *JN_constant*: " $(\sqcup i \in I. c) = (\text{if } I = \{\} \text{ then } SKIP \text{ else } c)$ "

<proof>

lemma *JN_Join_distrib*:

" $(\sqcup i \in I. F \ i \sqcup G \ i) = (\sqcup i \in I. F \ i) \sqcup (\sqcup i \in I. G \ i)$ "

<proof>

lemma *JN_Join_miniscope*:

" $i \in I \implies (\sqcup i \in I. F \ i \sqcup G) = ((\sqcup i \in I. F \ i) \sqcup G)$ "

<proof>

lemma *JN_Join_diff*: " $i \in I \implies F \ i \sqcup JOIN (I - \{i\}) F = JOIN I F$ "

<proof>

7.7 Safety: co, stable, FP

lemma *JN_constrains*:

" $i \in I \implies (\sqcup i \in I. F \ i) \in A \text{ co } B = (\forall i \in I. F \ i \in A \text{ co } B)$ "

<proof>

lemma *Join_constrains [simp]*:
 "(F⊔G ∈ A co B) = (F ∈ A co B & G ∈ A co B)"
 ⟨proof⟩

lemma *Join_unless [simp]*:
 "(F⊔G ∈ A unless B) = (F ∈ A unless B & G ∈ A unless B)"
 ⟨proof⟩

lemma *Join_constrains_weaken*:
 "[| F ∈ A co A'; G ∈ B co B' |]
 ==> F⊔G ∈ (A ∩ B) co (A' ∪ B')"
 ⟨proof⟩

lemma *JN_constrains_weaken*:
 "[| ∀i ∈ I. F i ∈ A i co A' i; i ∈ I |]
 ==> (⊔i ∈ I. F i) ∈ (∩i ∈ I. A i) co (⊔i ∈ I. A' i)"
 ⟨proof⟩

lemma *JN_stable*: "(⊔i ∈ I. F i) ∈ stable A = (∀i ∈ I. F i ∈ stable A)"
 ⟨proof⟩

lemma *invariant_JN_I*:
 "[| !!i. i ∈ I ==> F i ∈ invariant A; i ∈ I |]
 ==> (⊔i ∈ I. F i) ∈ invariant A"
 ⟨proof⟩

lemma *Join_stable [simp]*:
 "(F⊔G ∈ stable A) =
 (F ∈ stable A & G ∈ stable A)"
 ⟨proof⟩

lemma *Join_increasing [simp]*:
 "(F⊔G ∈ increasing f) =
 (F ∈ increasing f & G ∈ increasing f)"
 ⟨proof⟩

lemma *invariant_JoinI*:
 "[| F ∈ invariant A; G ∈ invariant A |]
 ==> F⊔G ∈ invariant A"
 ⟨proof⟩

lemma *FP_JN*: "FP (⊔i ∈ I. F i) = (∩i ∈ I. FP (F i))"
 ⟨proof⟩

7.8 Progress: transient, ensures

lemma *JN_transient*:
 "i ∈ I ==>
 (⊔i ∈ I. F i) ∈ transient A = (∃i ∈ I. F i ∈ transient A)"
 ⟨proof⟩

lemma *Join_transient [simp]:*

" $F \sqcup G \in \text{transient } A =$
 $(F \in \text{transient } A \mid G \in \text{transient } A)$ "

<proof>

lemma *Join_transient_I1:* " $F \in \text{transient } A \implies F \sqcup G \in \text{transient } A$ "

<proof>

lemma *Join_transient_I2:* " $G \in \text{transient } A \implies F \sqcup G \in \text{transient } A$ "

<proof>

lemma *JN_ensures:*

" $i \in I \implies$
 $(\bigsqcup_{i \in I}. F \ i) \in A \text{ ensures } B =$
 $((\forall i \in I. F \ i \in (A-B) \text{ co } (A \cup B)) \ \& \ (\exists i \in I. F \ i \in A \text{ ensures } B))$ "

<proof>

lemma *Join_ensures:*

" $F \sqcup G \in A \text{ ensures } B =$
 $(F \in (A-B) \text{ co } (A \cup B) \ \& \ G \in (A-B) \text{ co } (A \cup B) \ \&$
 $(F \in \text{transient } (A-B) \mid G \in \text{transient } (A-B)))$ "

<proof>

lemma *stable_Join_constrains:*

" $[| F \in \text{stable } A; \ G \in A \text{ co } A' \ |] \implies F \sqcup G \in A \text{ co } A'$ "

<proof>

lemma *stable_Join_Always1:*

" $[| F \in \text{stable } A; \ G \in \text{invariant } A \ |] \implies F \sqcup G \in \text{Always } A$ "

<proof>

lemma *stable_Join_Always2:*

" $[| F \in \text{invariant } A; \ G \in \text{stable } A \ |] \implies F \sqcup G \in \text{Always } A$ "

<proof>

lemma *stable_Join_ensures1:*

" $[| F \in \text{stable } A; \ G \in A \text{ ensures } B \ |] \implies F \sqcup G \in A \text{ ensures } B$ "

<proof>

lemma *stable_Join_ensures2:*

" $[| F \in A \text{ ensures } B; \ G \in \text{stable } A \ |] \implies F \sqcup G \in A \text{ ensures } B$ "

<proof>

7.9 the ok and OK relations

lemma *ok_SKIP1 [iff]:* " $\text{SKIP ok } F$ "

<proof>

lemma `ok_SKIP2 [iff]: "F ok SKIP"`
`<proof>`

lemma `ok_Join_commute:`
`"(F ok G & (F⊔G) ok H) = (G ok H & F ok (G⊔H))"`
`<proof>`

lemma `ok_commute: "(F ok G) = (G ok F)"`
`<proof>`

lemmas `ok_sym = ok_commute [THEN iffD1]`

lemma `ok_iff_OK:`
`"OK {(0::int,F),(1,G),(2,H)} snd = (F ok G & (F⊔G) ok H)"`
`<proof>`

lemma `ok_Join_iff1 [iff]: "F ok (G⊔H) = (F ok G & F ok H)"`
`<proof>`

lemma `ok_Join_iff2 [iff]: "(G⊔H) ok F = (G ok F & H ok F)"`
`<proof>`

lemma `ok_Join_commute_I: "[| F ok G; (F⊔G) ok H |] ==> F ok (G⊔H)"`
`<proof>`

lemma `ok_JN_iff1 [iff]: "F ok (JOIN I G) = (∀ i ∈ I. F ok G i)"`
`<proof>`

lemma `ok_JN_iff2 [iff]: "(JOIN I G) ok F = (∀ i ∈ I. G i ok F)"`
`<proof>`

lemma `OK_iff_ok: "OK I F = (∀ i ∈ I. ∀ j ∈ I-{i}. (F i) ok (F j))"`
`<proof>`

lemma `OK_imp_ok: "[| OK I F; i ∈ I; j ∈ I; i ≠ j |] ==> (F i) ok (F j)"`
`<proof>`

7.10 Allowed

lemma `Allowed_SKIP [simp]: "Allowed SKIP = UNIV"`
`<proof>`

lemma `Allowed_Join [simp]: "Allowed (F⊔G) = Allowed F ∩ Allowed G"`
`<proof>`

lemma `Allowed_JN [simp]: "Allowed (JOIN I F) = (∩ i ∈ I. Allowed (F i))"`
`<proof>`

lemma `ok_iff_Allowed: "F ok G = (F ∈ Allowed G & G ∈ Allowed F)"`
`<proof>`

lemma `OK_iff_Allowed: "OK I F = (∀ i ∈ I. ∀ j ∈ I-{i}. F i ∈ Allowed(F j))"`
`<proof>`

7.11 *safety_prop*, for reasoning about given instances of "ok"**lemma** *safety_prop_Acts_iff*:

"*safety_prop* $X \implies (\text{Acts } G \subseteq \text{insert Id } (\bigcup (\text{Acts } ' X))) = (G \in X)$ "
 <proof>

lemma *safety_prop_AllowedActs_iff_Allowed*:

"*safety_prop* $X \implies (\bigcup (\text{Acts } ' X) \subseteq \text{AllowedActs } F) = (X \subseteq \text{Allowed } F)$ "
 <proof>

lemma *Allowed_eq*:

"*safety_prop* $X \implies \text{Allowed } (\text{mk_program } (\text{init}, \text{acts}, \bigcup (\text{Acts } ' X))) = X$ "
 <proof>

lemma *safety_prop_constrains [iff]*: "*safety_prop* $(A \text{ co } B) = (A \subseteq B)$ "

<proof>

lemma *safety_prop_stable [iff]*: "*safety_prop* $(\text{stable } A)$ "

<proof>

lemma *safety_prop_Int [simp]*:

"*safety_prop* $X \implies \text{safety_prop } Y \implies \text{safety_prop } (X \cap Y)$ "
 <proof>

lemma *safety_prop_INTER [simp]*:

" $(\bigwedge i. i \in I \implies \text{safety_prop } (X i)) \implies \text{safety_prop } (\bigcap_{i \in I} X i)$ "
 <proof>

lemma *safety_prop_INTER1 [simp]*:

" $(\bigwedge i. \text{safety_prop } (X i)) \implies \text{safety_prop } (\bigcap i. X i)$ "
 <proof>

lemma *def_prg_Allowed*:

" $[| F = \text{mk_program } (\text{init}, \text{acts}, \bigcup (\text{Acts } ' X)) ; \text{safety_prop } X |]$
 $\implies \text{Allowed } F = X$ "
 <proof>

lemma *Allowed_totalize [simp]*: "*Allowed* $(\text{totalize } F) = \text{Allowed } F$ "

<proof>

lemma *def_total_prg_Allowed*:

" $[| F = \text{mk_total_program } (\text{init}, \text{acts}, \bigcup (\text{Acts } ' X)) ; \text{safety_prop } X |]$
 $\implies \text{Allowed } F = X$ "
 <proof>

lemma *def_UNION_ok_iff*:

" $[| F = \text{mk_program}(\text{init}, \text{acts}, \bigcup (\text{Acts } ' X)); \text{safety_prop } X |]$
 $\implies F \text{ ok } G = (G \in X \ \& \ \text{acts} \subseteq \text{AllowedActs } G)$ "
 <proof>

The union of two total programs is total.

```

lemma totalize_Join: "totalize F ⊔ totalize G = totalize (F ⊔ G)"
  ⟨proof⟩

lemma all_total_Join: "[|all_total F; all_total G|] ==> all_total (F ⊔ G)"
  ⟨proof⟩

lemma totalize_JN: "(⊔ i ∈ I. totalize (F i)) = totalize(⊔ i ∈ I. F i)"
  ⟨proof⟩

lemma all_total_JN: "(!!i. i ∈ I ==> all_total (F i)) ==> all_total(⊔ i ∈ I.
  F i)"
  ⟨proof⟩

end

```

8 Composition: Basic Primitives

```

theory Comp
imports Union
begin

instantiation program :: (type) ord
begin

definition component_def: "F ≤ H ↔ (∃ G. F ⊔ G = H)"

definition strict_component_def: "F < (H :: 'a program) ↔ (F ≤ H & F ≠ H)"

instance ⟨proof⟩

end

definition component_of :: "'a program => 'a program => bool" (infixl "component'_of"
50)
  where "F component_of H == ∃ G. F ok G & F ⊔ G = H"

definition strict_component_of :: "'a program => 'a program => bool" (infixl "strict'_component'_of"
50)
  where "F strict_component_of H == F component_of H & F ≠ H"

definition preserves :: "('a => 'b) => 'a program set"
  where "preserves v == ⋂ z. stable {s. v s = z}"

definition localize :: "('a => 'b) => 'a program => 'a program" where
  "localize v F == mk_program(Init F, Acts F,
  AllowedActs F ∩ (⋃ G ∈ preserves v. Acts G))"

definition funPair :: "'a => 'b, 'a => 'c, 'a] => 'b * 'c"
  where "funPair f g == %x. (f x, g x)"

```

8.1 The component relation

```

lemma componentI: "H ≤ F | H ≤ G ==> H ≤ (F ⊔ G)"

```

<proof>

lemma *component_eq_subset*:

" $(F \leq G) =$
 $(\text{Init } G \subseteq \text{Init } F \ \& \ \text{Acts } F \subseteq \text{Acts } G \ \& \ \text{AllowedActs } G \subseteq \text{AllowedActs } F)$ "

<proof>

lemma *component_SKIP [iff]*: " $\text{SKIP} \leq F$ "

<proof>

lemma *component_refl [iff]*: " $F \leq (F \text{ :: 'a program})$ "

<proof>

lemma *SKIP_minimal*: " $F \leq \text{SKIP} \implies F = \text{SKIP}$ "

<proof>

lemma *component_Join1*: " $F \leq (F \sqcup G)$ "

<proof>

lemma *component_Join2*: " $G \leq (F \sqcup G)$ "

<proof>

lemma *Join_absorb1*: " $F \leq G \implies F \sqcup G = G$ "

<proof>

lemma *Join_absorb2*: " $G \leq F \implies F \sqcup G = F$ "

<proof>

lemma *JN_component_iff*: " $((\text{JOIN } I \ F) \leq H) = (\forall i \in I. F \ i \leq H)$ "

<proof>

lemma *component_JN*: " $i \in I \implies (F \ i) \leq (\bigsqcup_{i \in I. (F \ i)})$ "

<proof>

lemma *component_trans*: " $[| F \leq G; G \leq H |] \implies F \leq (H \text{ :: 'a program})$ "

<proof>

lemma *component_antisym*: " $[| F \leq G; G \leq F |] \implies F = (G \text{ :: 'a program})$ "

<proof>

lemma *Join_component_iff*: " $((F \sqcup G) \leq H) = (F \leq H \ \& \ G \leq H)$ "

<proof>

lemma *component_constrains*: " $[| F \leq G; G \in A \text{ co } B |] \implies F \in A \text{ co } B$ "

<proof>

lemma *component_stable*: " $[| F \leq G; G \in \text{stable } A |] \implies F \in \text{stable } A$ "

<proof>

lemmas *program_less_le = strict_component_def*

8.2 The preserves property

```

lemma preservesI: "(!!z. F ∈ stable {s. v s = z}) ==> F ∈ preserves v"
⟨proof⟩

lemma preserves_imp_eq:
  "[| F ∈ preserves v; act ∈ Acts F; (s,s') ∈ act |] ==> v s = v s'"
⟨proof⟩

lemma Join_preserves [iff]:
  "(F ⊔ G ∈ preserves v) = (F ∈ preserves v & G ∈ preserves v)"
⟨proof⟩

lemma JN_preserves [iff]:
  "(JOIN I F ∈ preserves v) = (∀ i ∈ I. F i ∈ preserves v)"
⟨proof⟩

lemma SKIP_preserves [iff]: "SKIP ∈ preserves v"
⟨proof⟩

lemma funPair_apply [simp]: "(funPair f g) x = (f x, g x)"
⟨proof⟩

lemma preserves_funPair: "preserves (funPair v w) = preserves v ∩ preserves w"
⟨proof⟩

declare preserves_funPair [THEN eqset_imp_iff, iff]

lemma funPair_o_distrib: "(funPair f g) o h = funPair (f o h) (g o h)"
⟨proof⟩

lemma fst_o_funPair [simp]: "fst o (funPair f g) = f"
⟨proof⟩

lemma snd_o_funPair [simp]: "snd o (funPair f g) = g"
⟨proof⟩

lemma subset_preserves_o: "preserves v ⊆ preserves (w o v)"
⟨proof⟩

lemma preserves_subset_stable: "preserves v ⊆ stable {s. P (v s)}"
⟨proof⟩

lemma preserves_subset_increasing: "preserves v ⊆ increasing v"
⟨proof⟩

lemma preserves_id_subset_stable: "preserves id ⊆ stable A"
⟨proof⟩

```

```
lemma safety_prop_preserves [iff]: "safety_prop (preserves v)"
⟨proof⟩
```

```
lemma stable_localTo_stable2:
  "[| F ∈ stable {s. P (v s) (w s)};
    G ∈ preserves v; G ∈ preserves w |]
  ==> F ⊔ G ∈ stable {s. P (v s) (w s)}"
⟨proof⟩
```

```
lemma Increasing_preserves_Stable:
  "[| F ∈ stable {s. v s ≤ w s}; G ∈ preserves v; F ⊔ G ∈ Increasing w
  |]
  ==> F ⊔ G ∈ Stable {s. v s ≤ w s}"
⟨proof⟩
```

```
lemma component_of_imp_component: "F component_of H ==> F ≤ H"
⟨proof⟩
```

```
lemma component_of_refl [simp]: "F component_of F"
⟨proof⟩
```

```
lemma component_of_SKIP [simp]: "SKIP component_of F"
⟨proof⟩
```

```
lemma component_of_trans:
  "[| F component_of G; G component_of H |] ==> F component_of H"
⟨proof⟩
```

```
lemmas strict_component_of_eq = strict_component_of_def
```

```
lemma localize_Init_eq [simp]: "Init (localize v F) = Init F"
⟨proof⟩
```

```
lemma localize_Acts_eq [simp]: "Acts (localize v F) = Acts F"
⟨proof⟩
```

```
lemma localize_AllowedActs_eq [simp]:
  "AllowedActs (localize v F) = AllowedActs F ∩ (⋃ G ∈ preserves v. Acts
  G)"
⟨proof⟩
```

```
end
```

9 Guarantees Specifications

```
theory Guar
imports Comp
begin
```

```
instance program :: (type) order
  <proof>
```

Existential and Universal properties. I formalize the two-program case, proving equivalence with Chandy and Sanders's n-ary definitions

```
definition ex_prop :: "'a program set => bool" where
  "ex_prop X ==  $\forall F G. F \text{ ok } G \text{ -->} F \in X \mid G \in X \text{ -->} (F \sqcup G) \in X$ "
```

```
definition strict_ex_prop :: "'a program set => bool" where
  "strict_ex_prop X ==  $\forall F G. F \text{ ok } G \text{ -->} (F \in X \mid G \in X) = (F \sqcup G \in X)$ "
```

```
definition uv_prop :: "'a program set => bool" where
  "uv_prop X == SKIP  $\in X$  & ( $\forall F G. F \text{ ok } G \text{ -->} F \in X$  &  $G \in X \text{ -->} (F \sqcup G) \in X$ )"
```

```
definition strict_uv_prop :: "'a program set => bool" where
  "strict_uv_prop X ==
  SKIP  $\in X$  & ( $\forall F G. F \text{ ok } G \text{ -->} (F \in X$  &  $G \in X) = (F \sqcup G \in X)$ )"
```

Guarantees properties

```
definition guar :: "[ 'a program set, 'a program set ] => 'a program set" (infixl
  "guarantees" 55) where
```

```
  "X guarantees Y == {F.  $\forall G. F \text{ ok } G \text{ -->} F \sqcup G \in X \text{ -->} F \sqcup G \in Y$ }"
```

```
definition wg :: "[ 'a program, 'a program set ] => 'a program set" where
  "wg F Y ==  $\bigcup (\{X. F \in X \text{ guarantees } Y\})$ "
```

```
definition wx :: "('a program) set => ('a program)set" where
  "wx X ==  $\bigcup (\{Y. Y \subseteq X \text{ \& } \text{ex\_prop } Y\})$ "
```

```
definition welldef :: "'a program set" where
  "welldef == {F. Init F  $\neq \{\}$ }"
```

```
definition refines :: "[ 'a program, 'a program, 'a program set ] => bool"
  ("(3_ refines _ wrt _)" [10,10,10] 10) where
  "G refines F wrt X ==
   $\forall H. (F \text{ ok } H \text{ \& } G \text{ ok } H \text{ \& } F \sqcup H \in \text{welldef} \cap X) \text{ -->} (G \sqcup H \in \text{welldef} \cap X)$ "
```

```
definition iso_refines :: "[ 'a program, 'a program, 'a program set ] => bool"
  ("(3_ iso'_refines _ wrt _)" [10,10,10] 10) where
  "G iso_refines F wrt X ==
  F  $\in \text{welldef} \cap X \text{ -->} G \in \text{welldef} \cap X$ "
```



```

lemma OK_insert_iff:
  "(OK (insert i I) F) =
   (if i ∈ I then OK I F else OK I F & (F i ok JOIN I F))"
<proof>

```

9.1 Existential Properties

```

lemma ex1:
  assumes "ex_prop X" and "finite GG"
  shows "GG ∩ X ≠ {} ⇒ OK GG (%G. G) ⇒ (⋃ G ∈ GG. G) ∈ X"
<proof>

```

```

lemma ex2:
  "∀ GG. finite GG & GG ∩ X ≠ {} → OK GG (λG. G) → (⋃ G ∈ GG. G) ∈
X
 ⇒ ex_prop X"
<proof>

```

```

lemma ex_prop_finite:
  "ex_prop X =
  (∀ GG. finite GG & GG ∩ X ≠ {} & OK GG (%G. G) → (⋃ G ∈ GG. G) ∈ X)"
<proof>

```

```

lemma ex_prop_equiv:
  "ex_prop X = (∀ G. G ∈ X = (∀ H. (G component_of H) → H ∈ X))"
<proof>

```

9.2 Universal Properties

```

lemma uv1:
  assumes "uv_prop X"
  and "finite GG"
  and "GG ⊆ X"
  and "OK GG (%G. G)"
  shows "(⋃ G ∈ GG. G) ∈ X"
<proof>

```

```

lemma uv2:
  "∀ GG. finite GG & GG ⊆ X & OK GG (%G. G) → (⋃ G ∈ GG. G) ∈ X
 ⇒ uv_prop X"
<proof>

```

```

lemma uv_prop_finite:
  "uv_prop X =
  (∀ GG. finite GG ∧ GG ⊆ X ∧ OK GG (λG. G) → (⋃ G ∈ GG. G) ∈ X)"
<proof>

```

9.3 Guarantees

lemma *guaranteesI*:

"(!!G. [| F ok G; F ⊔ G ∈ X |] ==> F ⊔ G ∈ Y) ==> F ∈ X guarantees Y"
 <proof>

lemma *guaranteesD*:

"[| F ∈ X guarantees Y; F ok G; F ⊔ G ∈ X |] ==> F ⊔ G ∈ Y"
 <proof>

lemma *component_guaranteesD*:

"[| F ∈ X guarantees Y; F ⊔ G = H; H ∈ X; F ok G |] ==> H ∈ Y"
 <proof>

lemma *guarantees_weaken*:

"[| F ∈ X guarantees X'; Y ⊆ X; X' ⊆ Y' |] ==> F ∈ Y guarantees Y'"
 <proof>

lemma *subset_imp_guarantees_UNIV*: "X ⊆ Y ==> X guarantees Y = UNIV"
 <proof>

lemma *subset_imp_guarantees*: "X ⊆ Y ==> F ∈ X guarantees Y"
 <proof>

lemma *ex_prop_imp*: "ex_prop Y ==> (Y = UNIV guarantees Y)"
 <proof>

lemma *guarantees_imp*: "(Y = UNIV guarantees Y) ==> ex_prop(Y)"
 <proof>

lemma *ex_prop_equiv2*: "(ex_prop Y) = (Y = UNIV guarantees Y)"
 <proof>

9.4 Distributive Laws. Re-Orient to Perform Miniscoping

lemma *guarantees_UN_left*:

"(⋃ i ∈ I. X i) guarantees Y = (⋂ i ∈ I. X i guarantees Y)"
 <proof>

lemma *guarantees_Un_left*:

"(X ∪ Y) guarantees Z = (X guarantees Z) ∩ (Y guarantees Z)"
 <proof>

lemma *guarantees_INT_right*:

"X guarantees (⋂ i ∈ I. Y i) = (⋂ i ∈ I. X guarantees Y i)"
 <proof>

lemma *guarantees_Int_right*:

"Z guarantees (X ∩ Y) = (Z guarantees X) ∩ (Z guarantees Y)"
 <proof>

lemma *guarantees_Int_right_I*:

"[| F ∈ Z guarantees X; F ∈ Z guarantees Y |]
 ==> F ∈ Z guarantees (X ∩ Y)"

⟨proof⟩

lemma *guarantees_INT_right_iff*:

"(F ∈ X guarantees (∩ (Y ' I))) = (∀ i ∈ I. F ∈ X guarantees (Y i))"

⟨proof⟩

lemma *shunting*: "(X guarantees Y) = (UNIV guarantees (-X ∪ Y))"

⟨proof⟩

lemma *contrapositive*: "(X guarantees Y) = -Y guarantees -X"

⟨proof⟩

lemma *combining1*:

"[| F ∈ V guarantees X; F ∈ (X ∩ Y) guarantees Z |]
 ==> F ∈ (V ∩ Y) guarantees Z"

⟨proof⟩

lemma *combining2*:

"[| F ∈ V guarantees (X ∪ Y); F ∈ Y guarantees Z |]
 ==> F ∈ V guarantees (X ∪ Z)"

⟨proof⟩

lemma *all_guarantees*:

"∀ i ∈ I. F ∈ X guarantees (Y i) ==> F ∈ X guarantees (∩ i ∈ I. Y i)"

⟨proof⟩

lemma *ex_guarantees*:

"∃ i ∈ I. F ∈ X guarantees (Y i) ==> F ∈ X guarantees (∪ i ∈ I. Y i)"

⟨proof⟩

9.5 Guarantees: Additional Laws (by lcp)

lemma *guarantees_Join_Int*:

"[| F ∈ U guarantees V; G ∈ X guarantees Y; F ok G |]
 ==> F ⊔ G ∈ (U ∩ X) guarantees (V ∩ Y)"

⟨proof⟩

lemma *guarantees_Join_Un*:

"[| F ∈ U guarantees V; G ∈ X guarantees Y; F ok G |]
 ==> F ⊔ G ∈ (U ∪ X) guarantees (V ∪ Y)"

⟨proof⟩

lemma *guarantees_JN_INT*:

"[| ∀ i ∈ I. F i ∈ X i guarantees Y i; OK I F |]
 ==> (JOIN I F) ∈ (∩ (X ' I)) guarantees (∩ (Y ' I))"

<proof>

lemma *guarantees_JN_UN:*

"[| $\forall i \in I. F\ i \in X\ i$ guarantees $Y\ i$; $OK\ I\ F$ |]"
 $\implies (JOIN\ I\ F) \in (\bigcup (X\ 'I))$ guarantees $(\bigcup (Y\ 'I))$ "

<proof>

9.6 Guarantees Laws for Breaking Down the Program (by lcp)

lemma *guarantees_Join_I1:*

"[| $F \in X$ guarantees Y ; $F\ ok\ G$ |]" $\implies F \sqcup G \in X$ guarantees Y "

<proof>

lemma *guarantees_Join_I2:*

"[| $G \in X$ guarantees Y ; $F\ ok\ G$ |]" $\implies F \sqcup G \in X$ guarantees Y "

<proof>

lemma *guarantees_JN_I:*

"[| $i \in I$; $F\ i \in X$ guarantees Y ; $OK\ I\ F$ |]"
 $\implies (\bigcup i \in I. (F\ i)) \in X$ guarantees Y "

<proof>

lemma *Join_welldef_D1:* " $F \sqcup G \in welldef \implies F \in welldef$ "

<proof>

lemma *Join_welldef_D2:* " $F \sqcup G \in welldef \implies G \in welldef$ "

<proof>

lemma *refines_refl:* " F refines F wrt X "

<proof>

lemma *refines_trans:*

"[| H refines G wrt X ; G refines F wrt X |]" $\implies H$ refines F wrt X "

<proof>

lemma *strict_ex_refine_lemma:*

"*strict_ex_prop* X
 $\implies (\forall H. F\ ok\ H \ \&\ G\ ok\ H \ \&\ F \sqcup H \in X \implies G \sqcup H \in X)$
 $= (F \in X \implies G \in X)$ "

<proof>

lemma *strict_ex_refine_lemma_v:*

"*strict_ex_prop* X
 $\implies (\forall H. F\ ok\ H \ \&\ G\ ok\ H \ \&\ F \sqcup H \in welldef \ \&\ F \sqcup H \in X \implies G \sqcup H \in X) =$

$(F \in welldef \cap X \implies G \in X)$ "

<proof>

lemma *ex_refinement_thm*:

"[| strict_ex_prop X;
 $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \cap X \ \rightarrow G \sqcup H \in \text{welldef} \]$
 $\implies (G \text{ refines } F \text{ wrt } X) = (G \text{ iso_refines } F \text{ wrt } X)$ "

<proof>

lemma *strict_uv_refine_lemma*:

"strict_uv_prop X \implies
 $(\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in X \ \rightarrow G \sqcup H \in X) = (F \in X \ \rightarrow G \in X)$ "

<proof>

lemma *strict_uv_refine_lemma_v*:

"strict_uv_prop X
 $\implies (\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \ \& \ F \sqcup H \in X \ \rightarrow G \sqcup H \in X) =$
 $(F \in \text{welldef} \cap X \ \rightarrow G \in X)$ "

<proof>

lemma *uv_refinement_thm*:

"[| strict_uv_prop X;
 $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \cap X \ \rightarrow$
 $G \sqcup H \in \text{welldef} \]$
 $\implies (G \text{ refines } F \text{ wrt } X) = (G \text{ iso_refines } F \text{ wrt } X)$ "

<proof>

lemma *guarantees_equiv*:

" $(F \in X \text{ guarantees } Y) = (\forall H. H \in X \ \rightarrow (F \text{ component_of } H \ \rightarrow H \in Y))$ "

<proof>

lemma *wg_weakest*: " $!!X. F \in (X \text{ guarantees } Y) \implies X \subseteq (\text{wg } F \ Y)$ "

<proof>

lemma *wg_guarantees*: " $F \in ((\text{wg } F \ Y) \text{ guarantees } Y)$ "

<proof>

lemma *wg_equiv*: " $(H \in \text{wg } F \ X) = (F \text{ component_of } H \ \rightarrow H \in X)$ "

<proof>

lemma *component_of_wg*: " $F \text{ component_of } H \implies (H \in \text{wg } F \ X) = (H \in X)$ "

<proof>

lemma *wg_finite*:

" $\forall FF. \text{finite } FF \ \& \ FF \cap X \neq \{\}$ $\rightarrow OK \ FF \ (\lambda F. F)$
 $\rightarrow (\forall F \in FF. ((\bigsqcup F \in FF. F) \in \text{wg } F \ X) = ((\bigsqcup F \in FF. F) \in X))$ "

<proof>

lemma *wg_ex_prop*: " $\text{ex_prop } X \implies (F \in X) = (\forall H. H \in \text{wg } F \ X)$ "

<proof>

lemma `wx_subset`: " $(wx\ X) \leq X$ "
 $\langle proof \rangle$

lemma `wx_ex_prop`: " $ex_prop\ (wx\ X)$ "
 $\langle proof \rangle$

lemma `wx_weakest`: " $\forall Z. Z \leq X \rightarrow ex_prop\ Z \rightarrow Z \subseteq wx\ X$ "
 $\langle proof \rangle$

lemma `wx'_ex_prop`: " $ex_prop\ \{F. \forall G. F\ ok\ G \rightarrow F \sqcup G \in X\}$ "
 $\langle proof \rangle$

Equivalence with the other definition of `wx`

lemma `wx_equiv`: " $wx\ X = \{F. \forall G. F\ ok\ G \rightarrow (F \sqcup G) \in X\}$ "
 $\langle proof \rangle$

Propositions 7 to 11 are about this second definition of `wx`. They are the same as the ones proved for the first definition of `wx`, by equivalence

lemma `guarantees_wx_eq`: " $(X\ guarantees\ Y) = wx(-X \cup Y)$ "
 $\langle proof \rangle$

lemma `stable_guarantees_Always`:
 $"Init\ F \subseteq A \implies F \in (stable\ A)\ guarantees\ (Always\ A)"$
 $\langle proof \rangle$

lemma `constrains_guarantees_leadsTo`:
 $"F \in transient\ A \implies F \in (A\ co\ A \cup B)\ guarantees\ (A\ leadsTo\ (B-A))"$
 $\langle proof \rangle$

end

10 Extending State Sets

theory `Extend` imports `Guar` begin

definition

`Restrict` :: " $['a\ set, ('a*'b)\ set] \Rightarrow ('a*'b)\ set$ "
`where` "`Restrict` $A\ r = r \cap (A \times UNIV)$ "

definition

`good_map` :: " $['a*'b \Rightarrow 'c] \Rightarrow bool$ "
`where` "`good_map` $h \longleftrightarrow surj\ h \ \&\ (\forall x\ y. fst\ (inv\ h\ (h\ (x,y))) = x)$ "

definition

`extend_set` :: " $['a*'b \Rightarrow 'c, 'a\ set] \Rightarrow 'c\ set$ "
`where` "`extend_set` $h\ A = h\ ` (A \times UNIV)$ "

definition

```
project_set :: "[a*b => 'c, 'c set] => 'a set"
where "project_set h C = {x.  $\exists y. h(x,y) \in C$ }"
```

definition

```
extend_act :: "[a*b => 'c, ('a*a) set] => ('c*c) set"
where "extend_act h = (%act.  $\bigcup (s,s') \in act. \bigcup y. \{h(s,y), h(s',y)\}$ )"
```

definition

```
project_act :: "[a*b => 'c, ('c*c) set] => ('a*a) set"
where "project_act h act = {(x,x').  $\exists y y'. (h(x,y), h(x',y')) \in act$ }"
```

definition

```
extend :: "[a*b => 'c, 'a program] => 'c program"
where "extend h F = mk_program (extend_set h (Init F),
                               extend_act h ' Acts F,
                               project_act h -' AllowedActs F)"
```

definition

```
project :: "[a*b => 'c, 'c set, 'c program] => 'a program"
where "project h C F =
      mk_program (project_set h (Init F),
                 project_act h ' Restrict C ' Acts F,
                 {act. Restrict (project_set h C) act  $\in$ 
                  project_act h ' Restrict C ' AllowedActs F})"
```

locale Extend =

```
fixes f      :: "'c => 'a"
and g      :: "'c => 'b"
and h      :: "a*b => 'c"
and slice :: "[c set, 'b] => 'a set"
assumes
  good_h: "good_map h"
defines f_def: "f z == fst (inv h z)"
and g_def: "g z == snd (inv h z)"
and slice_def: "slice Z y == {x. h(x,y)  $\in Z$ }"
```

10.1 Restrict

lemma *Restrict_iff* [iff]: " $((x,y) \in \text{Restrict } A \ r) = ((x,y) \in r \ \& \ x \in A)$ "
<proof>

lemma *Restrict_UNIV* [simp]: " $\text{Restrict UNIV} = \text{id}$ "
<proof>

lemma *Restrict_empty* [simp]: " $\text{Restrict } \{\} \ r = \{\}$ "
<proof>

lemma *Restrict_Int* [simp]: " $\text{Restrict } A \ (\text{Restrict } B \ r) = \text{Restrict } (A \cap B) \ r$ "
<proof>

lemma *Restrict_triv*: " $\text{Domain } r \subseteq A \implies \text{Restrict } A \ r = r$ "

<proof>

lemma *Restrict_subset*: "Restrict A r \subseteq r"

<proof>

lemma *Restrict_eq_mono*:

"[| A \subseteq B; Restrict B r = Restrict B s |]
 \implies Restrict A r = Restrict A s"

<proof>

lemma *Restrict_imageI*:

"[| s \in RR; Restrict A r = Restrict A s |]
 \implies Restrict A r \in Restrict A ' RR"

<proof>

lemma *Domain_Restrict [simp]*: "Domain (Restrict A r) = A \cap Domain r"

<proof>

lemma *Image_Restrict [simp]*: "(Restrict A r) `` B = r `` (A \cap B)"

<proof>

lemma *good_mapI*:

assumes *surj_h*: "surj h"
 and *prem*: "!! x x' y y'. h(x,y) = h(x',y') \implies x=x'"
 shows "good_map h"

<proof>

lemma *good_map_is_surj*: "good_map h \implies surj h"

<proof>

lemma *fst_inv_equalityI*:

assumes *surj_h*: "surj h"
 and *prem*: "!! x y. g (h(x,y)) = x"
 shows "fst (inv h z) = g z"

<proof>

10.2 Trivial properties of f, g, h

context *Extend*

begin

lemma *f_h_eq [simp]*: "f(h(x,y)) = x"

<proof>

lemma *h_inject1 [dest]*: "h(x,y) = h(x',y') \implies x=x'"

<proof>

lemma *h_f_g_equiv*: "h(f z, g z) == z"

<proof>

lemma *h_f_g_eq*: "h(f z, g z) = z"

<proof>


```

lemma split_extended_all:
  "(!!z. PROP P z) == (!!u y. PROP P (h (u, y)))"
<proof>

end

```

10.3 extend_set: basic properties

```

lemma project_set_iff [iff]:
  "(x ∈ project_set h C) = (∃y. h(x,y) ∈ C)"
<proof>

```

```

lemma extend_set_mono: "A ⊆ B ==> extend_set h A ⊆ extend_set h B"
<proof>

```

```

context Extend
begin

```

```

lemma mem_extend_set_iff [iff]: "z ∈ extend_set h A = (f z ∈ A)"
<proof>

```

```

lemma extend_set_strict_mono [iff]:
  "(extend_set h A ⊆ extend_set h B) = (A ⊆ B)"
<proof>

```

```

lemma (in -) extend_set_empty [simp]: "extend_set h {} = {}"
<proof>

```

```

lemma extend_set_eq_Collect: "extend_set h {s. P s} = {s. P(f s)}"
<proof>

```

```

lemma extend_set_sing: "extend_set h {x} = {s. f s = x}"
<proof>

```

```

lemma extend_set_inverse [simp]: "project_set h (extend_set h C) = C"
<proof>

```

```

lemma extend_set_project_set: "C ⊆ extend_set h (project_set h C)"
<proof>

```

```

lemma inj_extend_set: "inj (extend_set h)"
<proof>

```

```

lemma extend_set_UNIV_eq [simp]: "extend_set h UNIV = UNIV"
<proof>

```

10.4 project_set: basic properties

```

lemma project_set_eq: "project_set h C = f ` C"
<proof>

```

lemma *project_set_I*: " $\forall z. z \in C \implies f z \in \text{project_set } h C$ "
<proof>

10.5 More laws

lemma *project_set_extend_set_Int*: " $\text{project_set } h ((\text{extend_set } h A) \cap B) = A \cap (\text{project_set } h B)$ "
<proof>

lemma *project_set_extend_set_Un*: " $\text{project_set } h ((\text{extend_set } h A) \cup B) = A \cup (\text{project_set } h B)$ "
<proof>

lemma *(in -) project_set_Int_subset*:
" $\text{project_set } h (A \cap B) \subseteq (\text{project_set } h A) \cap (\text{project_set } h B)$ "
<proof>

lemma *extend_set_Un_distrib*: " $\text{extend_set } h (A \cup B) = \text{extend_set } h A \cup \text{extend_set } h B$ "
<proof>

lemma *extend_set_Int_distrib*: " $\text{extend_set } h (A \cap B) = \text{extend_set } h A \cap \text{extend_set } h B$ "
<proof>

lemma *extend_set_INT_distrib*: " $\text{extend_set } h (\bigcap (B \text{ ' } A)) = (\bigcap x \in A. \text{extend_set } h (B x))$ "
<proof>

lemma *extend_set_Diff_distrib*: " $\text{extend_set } h (A - B) = \text{extend_set } h A - \text{extend_set } h B$ "
<proof>

lemma *extend_set_Union*: " $\text{extend_set } h (\bigcup A) = (\bigcup X \in A. \text{extend_set } h X)$ "
<proof>

lemma *extend_set_subset_Compl_eq*: " $(\text{extend_set } h A \subseteq - \text{extend_set } h B) = (A \subseteq - B)$ "
<proof>

10.6 extend_act

lemma *mem_extend_act_iff [iff]*: " $((h(s,y), h(s',y)) \in \text{extend_act } h \text{ act}) = ((s, s') \in \text{act})$ "
<proof>

lemma *extend_act_D*: " $(z, z') \in \text{extend_act } h \text{ act} \implies (f z, f z') \in \text{act}$ "
<proof>

lemma *extend_act_inverse [simp]*: " $\text{project_act } h (\text{extend_act } h \text{ act}) = \text{act}$ "
<proof>

```

lemma project_act_extend_act_restrict [simp]:
  "project_act h (Restrict C (extend_act h act)) =
   Restrict (project_set h C) act"
  <proof>

lemma subset_extend_act_D: "act'  $\subseteq$  extend_act h act ==> project_act h act'
 $\subseteq$  act"
  <proof>

lemma inj_extend_act: "inj (extend_act h)"
  <proof>

lemma extend_act_Image [simp]:
  "extend_act h act `` (extend_set h A) = extend_set h (act `` A)"
  <proof>

lemma extend_act_strict_mono [iff]:
  "(extend_act h act'  $\subseteq$  extend_act h act) = (act' <= act)"
  <proof>

lemma [iff]: "(extend_act h act = extend_act h act') = (act = act')"
  <proof>

lemma (in -) Domain_extend_act:
  "Domain (extend_act h act) = extend_set h (Domain act)"
  <proof>

lemma extend_act_Id [simp]: "extend_act h Id = Id"
  <proof>

lemma project_act_I: "!!z z'. (z, z')  $\in$  act ==> (f z, f z')  $\in$  project_act
h act"
  <proof>

lemma project_act_Id [simp]: "project_act h Id = Id"
  <proof>

lemma Domain_project_act: "Domain (project_act h act) = project_set h (Domain
act)"
  <proof>

```

10.7 extend

Basic properties

```

lemma (in -) Init_extend [simp]:
  "Init (extend h F) = extend_set h (Init F)"
  <proof>

lemma (in -) Init_project [simp]:
  "Init (project h C F) = project_set h (Init F)"
  <proof>

lemma Acts_extend [simp]: "Acts (extend h F) = (extend_act h ` Acts F)"

```

<proof>

lemma *AllowedActs_extend [simp]:*

"AllowedActs (extend h F) = project_act h -' AllowedActs F"

<proof>

lemma (in -) *Acts_project [simp]:*

"Acts(project h C F) = insert Id (project_act h ' Restrict C ' Acts F)"

<proof>

lemma *AllowedActs_project [simp]:*

"AllowedActs(project h C F) =
 {act. Restrict (project_set h C) act
 ∈ project_act h ' Restrict C ' AllowedActs F}"

<proof>

lemma *Allowed_extend:* "Allowed (extend h F) = project h UNIV -' Allowed F"

<proof>

lemma *extend_SKIP [simp]:* "extend h SKIP = SKIP"

<proof>

lemma (in -) *project_set_UNIV [simp]:* "project_set h UNIV = UNIV"

<proof>

lemma (in -) *project_set_Union:* "project_set h ($\bigcup A$) = ($\bigcup X \in A.$ project_set h X)"

<proof>

lemma (in -) *project_act.Restrict_subset:*

"project_act h (Restrict C act) \subseteq Restrict (project_set h C) (project_act h act)"

<proof>

lemma *project_act.Restrict_Id_eq:* "project_act h (Restrict C Id) = Restrict (project_set h C) Id"

<proof>

lemma *project_extend_eq:*

"project h C (extend h F) =
 mk_program (Init F, Restrict (project_set h C) ' Acts F,
 {act. Restrict (project_set h C) act
 ∈ project_act h ' Restrict C '
 (project_act h -' AllowedActs F)})"

<proof>

lemma *extend_inverse [simp]:*

"project h UNIV (extend h F) = F"

<proof>

lemma *inj_extend:* "inj (extend h)"

<proof>

lemma *extend_Join [simp]*: "extend h (F \sqcup G) = extend h F \sqcup extend h G"
 <proof>

lemma *extend_JN [simp]*: "extend h (JOIN I F) = (\bigsqcup i \in I. extend h (F i))"
 <proof>

lemma *extend_mono*: "F \leq G ==> extend h F \leq extend h G"
 <proof>

lemma *project_mono*: "F \leq G ==> project h C F \leq project h C G"
 <proof>

lemma *all_total_extend*: "all_total F ==> all_total (extend h F)"
 <proof>

10.8 Safety: co, stable

lemma *extend_constrains*:
 "(extend h F \in (extend_set h A) co (extend_set h B)) =
 (F \in A co B)"
 <proof>

lemma *extend_stable*:
 "(extend h F \in stable (extend_set h A)) = (F \in stable A)"
 <proof>

lemma *extend_invariant*:
 "(extend h F \in invariant (extend_set h A)) = (F \in invariant A)"
 <proof>

lemma *extend_constrains_project_set*:
 "extend h F \in A co B ==> F \in (project_set h A) co (project_set h B)"
 <proof>

lemma *extend_stable_project_set*:
 "extend h F \in stable A ==> F \in stable (project_set h A)"
 <proof>

10.9 Weak safety primitives: Co, Stable

lemma *reachable_extend_f*: "p \in reachable (extend h F) ==> f p \in reachable F"
 <proof>

lemma *h_reachable_extend*: "h(s,y) \in reachable (extend h F) ==> s \in reachable F"
 <proof>

lemma *reachable_extend_eq*: "reachable (extend h F) = extend_set h (reachable F)"

<proof>

lemma *extend_Constrains*:

"(extend h F ∈ (extend_set h A) Co (extend_set h B)) =
(F ∈ A Co B)"

<proof>

lemma *extend_Stable*: "(extend h F ∈ Stable (extend_set h A)) = (F ∈ Stable A)"

<proof>

lemma *extend_Always*: "(extend h F ∈ Always (extend_set h A)) = (F ∈ Always A)"

<proof>

lemma (in -) *project_act_mono*:

"D ⊆ C ==>

project_act h (Restrict D act) ⊆ project_act h (Restrict C act)"

<proof>

lemma *project_constrains_mono*:

"[| D ⊆ C; project h C F ∈ A co B |] ==> project h D F ∈ A co B"

<proof>

lemma *project_stable_mono*:

"[| D ⊆ C; project h C F ∈ stable A |] ==> project h D F ∈ stable A"

<proof>

lemma *project_constrains*:

"(project h C F ∈ A co B) =

(F ∈ (C ∩ extend_set h A) co (extend_set h B) & A ⊆ B)"

<proof>

lemma *project_stable*: "(project h UNIV F ∈ stable A) = (F ∈ stable (extend_set h A))"

<proof>

lemma *project_stable_I*: "F ∈ stable (extend_set h A) ==> project h C F ∈ stable A"

<proof>

lemma *Int_extend_set_lemma*:

"A ∩ extend_set h ((project_set h A) ∩ B) = A ∩ extend_set h B"

<proof>

lemma *project_constrains_project_set*:

"G ∈ C co B ==> project h C G ∈ project_set h C co project_set h B"

<proof>

lemma *project_stable_project_set*:
 " $G \in \text{stable } C \implies \text{project } h \ C \ G \in \text{stable } (\text{project_set } h \ C)$ "
<proof>

10.10 Progress: transient, ensures

lemma *extend_transient*:
 " $(\text{extend } h \ F \in \text{transient } (\text{extend_set } h \ A)) = (F \in \text{transient } A)$ "
<proof>

lemma *extend_ensures*:
 " $(\text{extend } h \ F \in (\text{extend_set } h \ A) \ \text{ensures } (\text{extend_set } h \ B)) =$
 $(F \in A \ \text{ensures } B)$ "
<proof>

lemma *leadsTo_imp_extend_leadsTo*:
 " $F \in A \ \text{leadsTo } B$
 $\implies \text{extend } h \ F \in (\text{extend_set } h \ A) \ \text{leadsTo } (\text{extend_set } h \ B)$ "
<proof>

10.11 Proving the converse takes some doing!

lemma *slice_iff [iff]*: " $(x \in \text{slice } C \ y) = (h(x,y) \in C)$ "
<proof>

lemma *slice_Union*: " $\text{slice } (\bigcup S) \ y = (\bigcup x \in S. \ \text{slice } x \ y)$ "
<proof>

lemma *slice_extend_set*: " $\text{slice } (\text{extend_set } h \ A) \ y = A$ "
<proof>

lemma *project_set_is_UN_slice*: " $\text{project_set } h \ A = (\bigcup y. \ \text{slice } A \ y)$ "
<proof>

lemma *extend_transient_slice*:
 " $\text{extend } h \ F \in \text{transient } A \implies F \in \text{transient } (\text{slice } A \ y)$ "
<proof>

lemma *extend_constrains_slice*:
 " $\text{extend } h \ F \in A \ \text{co } B \implies F \in (\text{slice } A \ y) \ \text{co } (\text{slice } B \ y)$ "
<proof>

lemma *extend_ensures_slice*:
 " $\text{extend } h \ F \in A \ \text{ensures } B \implies F \in (\text{slice } A \ y) \ \text{ensures } (\text{project_set } h$
 $B)$ "
<proof>

lemma *leadsTo_slice_project_set*:
 " $\forall y. \ F \in (\text{slice } B \ y) \ \text{leadsTo } CU \implies F \in (\text{project_set } h \ B) \ \text{leadsTo } CU$ "
<proof>

```

lemma extend_leadsTo_slice [rule_format]:
  "extend h F ∈ AU leadsTo BU
   ==> ∀y. F ∈ (slice AU y) leadsTo (project_set h BU)"
⟨proof⟩

lemma extend_leadsTo:
  "(extend h F ∈ (extend_set h A) leadsTo (extend_set h B)) =
   (F ∈ A leadsTo B)"
⟨proof⟩

lemma extend_LeadsTo:
  "(extend h F ∈ (extend_set h A) LeadsTo (extend_set h B)) =
   (F ∈ A LeadsTo B)"
⟨proof⟩

```

10.12 preserves

```

lemma project_preserves_I:
  "G ∈ preserves (v o f) ==> project h C G ∈ preserves v"
⟨proof⟩

lemma project_preserves_id_I:
  "G ∈ preserves f ==> project h C G ∈ preserves id"
⟨proof⟩

lemma extend_preserves:
  "(extend h G ∈ preserves (v o f)) = (G ∈ preserves v)"
⟨proof⟩

lemma inj_extend_preserves: "inj h ==> (extend h G ∈ preserves g)"
⟨proof⟩

```

10.13 Guarantees

```

lemma project_extend_Join: "project h UNIV ((extend h F)⊔G) = F⊔(project
h UNIV G)"
⟨proof⟩

lemma extend_Join_eq_extend_D:
  "(extend h F)⊔G = extend h H ==> H = F⊔(project h UNIV G)"
⟨proof⟩

lemma ok_extend_imp_ok_project: "extend h F ok G ==> F ok project h UNIV
G"
⟨proof⟩

lemma ok_extend_iff: "(extend h F ok extend h G) = (F ok G)"
⟨proof⟩

lemma OK_extend_iff: "OK I (%i. extend h (F i)) = (OK I F)"

```



```

⟨proof⟩

lemma guarantees_imp_extend_guarantees:
  "F ∈ X guarantees Y ==>
   extend h F ∈ (extend h ' X) guarantees (extend h ' Y)"
⟨proof⟩

lemma extend_guarantees_imp_guarantees:
  "extend h F ∈ (extend h ' X) guarantees (extend h ' Y)
   ==> F ∈ X guarantees Y"
⟨proof⟩

lemma extend_guarantees_eq:
  "(extend h F ∈ (extend h ' X) guarantees (extend h ' Y)) =
   (F ∈ X guarantees Y)"
  ⟨proof⟩

end

end

```

11 Renaming of State Sets

```
theory Rename imports Extend begin
```

```
definition rename :: "[ 'a => 'b, 'a program ] => 'b program" where
  "rename h == extend (%(x,u)::unit). h x"
```

```
declare image_inv_f_f [simp] image_f_inv_f [simp]
```

```
declare Extend.intro [simp,intro]
```

```
lemma good_map_bij [simp,intro]: "bij h ==> good_map (%(x,u). h x)"
⟨proof⟩
```

```
lemma fst_o_inv_eq_inv: "bij h ==> fst (inv (%(x,u). h x) s) = inv h s"
⟨proof⟩
```

```
lemma mem_rename_set_iff: "bij h ==> z ∈ h'A = (inv h z ∈ A)"
⟨proof⟩
```

```
lemma extend_set_eq_image [simp]: "extend_set (%(x,u). h x) A = h'A"
⟨proof⟩
```

```
lemma Init_rename [simp]: "Init (rename h F) = h'(Init F)"
⟨proof⟩
```

11.1 inverse properties

```
lemma extend_set_inv:
  "bij h
   ==> extend_set (%(x,u)::'c). inv h x) = project_set (%(x,u)::'c). h x)"

```

<proof>

lemma *bij_extend_act_eq_project_act*: "bij h
 $\implies \text{extend_act } (\% (x, u :: 'c). h x) = \text{project_act } (\% (x, u :: 'c). \text{inv } h x)$ "
<proof>

lemma *bij_extend_act*: "bij h \implies bij (extend_act (%(x,u::'c). h x))"
<proof>

lemma *bij_project_act*: "bij h \implies bij (project_act (%(x,u::'c). h x))"
<proof>

lemma *bij_inv_project_act_eq*: "bij h \implies inv (project_act (%(x,u::'c). inv h x)) =
 $\text{project_act } (\% (x, u :: 'c). h x)$ "
<proof>

lemma *extend_inv*: "bij h
 $\implies \text{extend } (\% (x, u :: 'c). \text{inv } h x) = \text{project } (\% (x, u :: 'c). h x) \text{ UNIV}$ "
<proof>

lemma *rename_inv_rename [simp]*: "bij h \implies rename (inv h) (rename h F) =
F"
<proof>

lemma *rename_rename_inv [simp]*: "bij h \implies rename h (rename (inv h) F) =
F"
<proof>

lemma *rename_inv_eq*: "bij h \implies rename (inv h) = inv (rename h)"
<proof>

lemma *bij_extend*: "bij h \implies bij (extend (%(x,u::'c). h x))"
<proof>

lemma *bij_project*: "bij h \implies bij (project (%(x,u::'c). h x) UNIV)"
<proof>

lemma *inv_project_eq*:
"bij h
 $\implies \text{inv } (\text{project } (\% (x, u :: 'c). h x) \text{ UNIV}) = \text{extend } (\% (x, u :: 'c). h x)$ "
<proof>

lemma *Allowed_rename [simp]*:
"bij h \implies Allowed (rename h F) = rename h ' Allowed F"
<proof>

lemma *bij_rename*: "bij h \implies bij (rename h)"
<proof>

lemmas *surj_rename* = bij_rename [THEN bij_is_surj]

lemma *inj_rename_imp_inj*: "inj (rename h) ==> inj h"
 <proof>

lemma *surj_rename_imp_surj*: "surj (rename h) ==> surj h"
 <proof>

lemma *bij_rename_imp_bij*: "bij (rename h) ==> bij h"
 <proof>

lemma *bij_rename_iff [simp]*: "bij (rename h) = bij h"
 <proof>

11.2 the lattice operations

lemma *rename_SKIP [simp]*: "bij h ==> rename h SKIP = SKIP"
 <proof>

lemma *rename_Join [simp]*:
 "bij h ==> rename h (F \sqcup G) = rename h F \sqcup rename h G"
 <proof>

lemma *rename_JN [simp]*:
 "bij h ==> rename h (JOIN I F) = (\bigsqcup i \in I. rename h (F i))"
 <proof>

11.3 Strong Safety: co, stable

lemma *rename_constrains*:
 "bij h ==> (rename h F \in (h'A) co (h'B)) = (F \in A co B)"
 <proof>

lemma *rename_stable*:
 "bij h ==> (rename h F \in stable (h'A)) = (F \in stable A)"
 <proof>

lemma *rename_invariant*:
 "bij h ==> (rename h F \in invariant (h'A)) = (F \in invariant A)"
 <proof>

lemma *rename_increasing*:
 "bij h ==> (rename h F \in increasing func) = (F \in increasing (func o h))"
 <proof>

11.4 Weak Safety: Co, Stable

lemma *reachable_rename_eq*:
 "bij h ==> reachable (rename h F) = h ' (reachable F)"
 <proof>

lemma *rename_Constrains*:
 "bij h ==> (rename h F \in (h'A) Co (h'B)) = (F \in A Co B)"
 <proof>

<proof>

11.6 "image" versions of the rules, for lifting "guarantees" properties

lemmas *bij_eq_rename = surj_rename [THEN surj_f_inv_f, symmetric]*

lemma *rename_image_constrains:*

"bij h ==> rename h ' (A co B) = (h ' A) co (h'B)"

<proof>

lemma *rename_image_stable: "bij h ==> rename h ' stable A = stable (h ' A)"*

<proof>

lemma *rename_image_increasing:*

"bij h ==> rename h ' increasing func = increasing (func o inv h)"

<proof>

lemma *rename_image_invariant:*

"bij h ==> rename h ' invariant A = invariant (h ' A)"

<proof>

lemma *rename_image_Constrains:*

"bij h ==> rename h ' (A Co B) = (h ' A) Co (h'B)"

<proof>

lemma *rename_image_preserves:*

"bij h ==> rename h ' preserves v = preserves (v o inv h)"

<proof>

lemma *rename_image_Stable:*

"bij h ==> rename h ' Stable A = Stable (h ' A)"

<proof>

lemma *rename_image_Increasing:*

"bij h ==> rename h ' Increasing func = Increasing (func o inv h)"

<proof>

lemma *rename_image_Always: "bij h ==> rename h ' Always A = Always (h ' A)"*

<proof>

lemma *rename_image_leadsTo:*

"bij h ==> rename h ' (A leadsTo B) = (h ' A) leadsTo (h'B)"

<proof>

lemma *rename_image_LeadsTo:*

"bij h ==> rename h ' (A LeadsTo B) = (h ' A) LeadsTo (h'B)"

<proof>

end

12 Replication of Components

theory *Lift_prog* imports *Rename* begin

```
definition insert_map :: "[nat, 'b, nat=>'b] => (nat=>'b)" where
  "insert_map i z f k == if k<i then f k
                        else if k=i then z
                        else f(k - 1)"
```

```
definition delete_map :: "[nat, nat=>'b] => (nat=>'b)" where
  "delete_map i g k == if k<i then g k else g (Suc k)"
```

```
definition lift_map :: "[nat, 'b * ((nat=>'b) * 'c)] => (nat=>'b) * 'c" where
  "lift_map i == %(s,(f,uu)). (insert_map i s f, uu)"
```

```
definition drop_map :: "[nat, (nat=>'b) * 'c] => 'b * ((nat=>'b) * 'c)" where
  "drop_map i == %(g, uu). (g i, (delete_map i g, uu))"
```

```
definition lift_set :: "[nat, ('b * ((nat=>'b) * 'c)) set] => ((nat=>'b) *
'c) set" where
  "lift_set i A == lift_map i ` A"
```

```
definition lift :: "[nat, ('b * ((nat=>'b) * 'c)) program] => ((nat=>'b) *
'c) program" where
  "lift i == rename (lift_map i)"
```

```
definition sub :: "[ 'a, 'a=>'b] => 'b" where
  "sub == %i f. f i"
```

```
declare insert_map_def [simp] delete_map_def [simp]
```

```
lemma insert_map_inverse: "delete_map i (insert_map i x f) = f"
<proof>
```

```
lemma insert_map_delete_map_eq: "(insert_map i x (delete_map i g)) = g(i:=x)"
<proof>
```

12.1 Injectiveness proof

```
lemma insert_map_inject1: "(insert_map i x f) = (insert_map i y g) ==> x=y"
<proof>
```

```
lemma insert_map_inject2: "(insert_map i x f) = (insert_map i y g) ==> f=g"
<proof>
```

```
lemma insert_map_inject':
  "(insert_map i x f) = (insert_map i y g) ==> x=y & f=g"
<proof>
```

```
lemmas insert_map_inject = insert_map_inject' [THEN conjE, elim!]
```

```

lemma lift_map_eq_iff [iff]:
  "(lift_map i (s, (f, uu)) = lift_map i' (s', (f', uu')))
   = (uu = uu' & insert_map i s f = insert_map i' s' f')"
<proof>

```

```

lemma drop_map_lift_map_eq [simp]: "!!s. drop_map i (lift_map i s) = s"
<proof>

```

```

lemma inj_lift_map: "inj (lift_map i)"
<proof>

```

12.2 Surjectiveness proof

```

lemma lift_map_drop_map_eq [simp]: "!!s. lift_map i (drop_map i s) = s"
<proof>

```

```

lemma drop_map_inject [dest!]: "(drop_map i s) = (drop_map i s') ==> s=s'"
<proof>

```

```

lemma surj_lift_map: "surj (lift_map i)"
<proof>

```

```

lemma bij_lift_map [iff]: "bij (lift_map i)"
<proof>

```

```

lemma inv_lift_map_eq [simp]: "inv (lift_map i) = drop_map i"
<proof>

```

```

lemma inv_drop_map_eq [simp]: "inv (drop_map i) = lift_map i"
<proof>

```

```

lemma bij_drop_map [iff]: "bij (drop_map i)"
<proof>

```

```

lemma sub_apply [simp]: "sub i f = f i"
<proof>

```

```

lemma all_total_lift: "all_total F ==> all_total (lift i F)"
<proof>

```

```

lemma insert_map_upd_same: "(insert_map i t f)(i := s) = insert_map i s f"
<proof>

```

```

lemma insert_map_upd:
  "(insert_map j t f)(i := s) =
   (if i=j then insert_map i s f
    else if i<j then insert_map j t (f(i:=s))
    else insert_map j t (f(i - Suc 0 := s)))"
<proof>

```

```

lemma insert_map_eq_diff:
  "[[ insert_map i s f = insert_map j t g; i≠j ]]

```

$\implies \exists g'. \text{insert_map } i \ s' \ f = \text{insert_map } j \ t \ g'$
 $\langle \text{proof} \rangle$

lemma *lift_map_eq_diff*:
 $"[| \text{lift_map } i \ (s, (f, uu)) = \text{lift_map } j \ (t, (g, vv)); \ i \neq j \ |]$
 $\implies \exists g'. \text{lift_map } i \ (s', (f, uu)) = \text{lift_map } j \ (t, (g', vv))"$
 $\langle \text{proof} \rangle$

12.3 The Operator *lift_set*

lemma *lift_set_empty [simp]*: " $\text{lift_set } i \ \{\} = \{\}$ "
 $\langle \text{proof} \rangle$

lemma *lift_set_iff*: " $(\text{lift_map } i \ x \in \text{lift_set } i \ A) = (x \in A)$ "
 $\langle \text{proof} \rangle$

lemma *lift_set_iff2 [iff]*:
 $"((f, uu) \in \text{lift_set } i \ A) = ((f \ i, (\text{delete_map } i \ f, uu)) \in A)"$
 $\langle \text{proof} \rangle$

lemma *lift_set_mono*: " $A \subseteq B \implies \text{lift_set } i \ A \subseteq \text{lift_set } i \ B$ "
 $\langle \text{proof} \rangle$

lemma *lift_set_Un_distrib*: " $\text{lift_set } i \ (A \cup B) = \text{lift_set } i \ A \cup \text{lift_set } i \ B$ "
 $\langle \text{proof} \rangle$

lemma *lift_set_Diff_distrib*: " $\text{lift_set } i \ (A - B) = \text{lift_set } i \ A - \text{lift_set } i \ B$ "
 $\langle \text{proof} \rangle$

12.4 The Lattice Operations

lemma *bij_lift [iff]*: " $\text{bij } (\text{lift } i)$ "
 $\langle \text{proof} \rangle$

lemma *lift_SKIP [simp]*: " $\text{lift } i \ \text{SKIP} = \text{SKIP}$ "
 $\langle \text{proof} \rangle$

lemma *lift_Join [simp]*: " $\text{lift } i \ (F \sqcup G) = \text{lift } i \ F \sqcup \text{lift } i \ G$ "
 $\langle \text{proof} \rangle$

lemma *lift_JN [simp]*: " $\text{lift } j \ (\text{JOIN } I \ F) = (\bigsqcup_{i \in I}. \text{lift } j \ (F \ i))"$
 $\langle \text{proof} \rangle$

12.5 Safety: constrains, stable, invariant

lemma *lift_constrains*:
 $"(\text{lift } i \ F \in (\text{lift_set } i \ A) \text{ co } (\text{lift_set } i \ B)) = (F \in A \text{ co } B)"$
 $\langle \text{proof} \rangle$

lemma *lift_stable*:


```
"(lift i F ∈ stable (lift_set i A)) = (F ∈ stable A)"
⟨proof⟩
```

```
lemma lift_invariant:
  "(lift i F ∈ invariant (lift_set i A)) = (F ∈ invariant A)"
⟨proof⟩
```

```
lemma lift_Constrains:
  "(lift i F ∈ (lift_set i A) Co (lift_set i B)) = (F ∈ A Co B)"
⟨proof⟩
```

```
lemma lift_Stable:
  "(lift i F ∈ Stable (lift_set i A)) = (F ∈ Stable A)"
⟨proof⟩
```

```
lemma lift_Always:
  "(lift i F ∈ Always (lift_set i A)) = (F ∈ Always A)"
⟨proof⟩
```

12.6 Progress: transient, ensures

```
lemma lift_transient:
  "(lift i F ∈ transient (lift_set i A)) = (F ∈ transient A)"
⟨proof⟩
```

```
lemma lift_ensures:
  "(lift i F ∈ (lift_set i A) ensures (lift_set i B)) =
  (F ∈ A ensures B)"
⟨proof⟩
```

```
lemma lift_leadsTo:
  "(lift i F ∈ (lift_set i A) leadsTo (lift_set i B)) =
  (F ∈ A leadsTo B)"
⟨proof⟩
```

```
lemma lift_LeadsTo:
  "(lift i F ∈ (lift_set i A) LeadsTo (lift_set i B)) =
  (F ∈ A LeadsTo B)"
⟨proof⟩
```

```
lemma lift_lift_guarantees_eq:
  "(lift i F ∈ (lift i ' X) guarantees (lift i ' Y)) =
  (F ∈ X guarantees Y)"
⟨proof⟩
```

```
lemma lift_guarantees_eq_lift_inv:
  "(lift i F ∈ X guarantees Y) =
  (F ∈ (rename (drop_map i) ' X) guarantees (rename (drop_map i) ' Y))"
⟨proof⟩
```

lemma *lift_preserves_snd_I*: "F ∈ preserves snd ==> lift i F ∈ preserves snd"

<proof>

lemma *delete_map_eqE'*:

"(delete_map i g) = (delete_map i g') ==> ∃x. g = g'(i:=x)"

<proof>

lemmas *delete_map_eqE* = *delete_map_eqE'* [THEN exE, elim!]

lemma *delete_map_neq_apply*:

"[| delete_map j g = delete_map j g'; i ≠ j |] ==> g i = g' i"

<proof>

lemma *vimage_o_fst_eq [simp]*: "(f o fst) -' A = (f-'A) × UNIV"

<proof>

lemma *vimage_sub_eq_lift_set [simp]*:

"(sub i -'A) × UNIV = lift_set i (A × UNIV)"

<proof>

lemma *mem_lift_act_iff [iff]*:

"((s,s') ∈ extend_act (%(x,u::unit). lift_map i x) act) =
((drop_map i s, drop_map i s') ∈ act)"

<proof>

lemma *preserves_snd_lift_stable*:

"[| F ∈ preserves snd; i ≠ j |]
==> lift j F ∈ stable (lift_set i (A × UNIV))"

<proof>

lemma *constrains_imp_lift_constrains*:

"[| F i ∈ (A × UNIV) co (B × UNIV);
F j ∈ preserves snd |]
==> lift j (F j) ∈ (lift_set i (A × UNIV)) co (lift_set i (B × UNIV))"

<proof>

lemma *lift_map_image_Times*:

"lift_map i ' (A × UNIV) =
(∪ s ∈ A. ∪ f. {insert_map i s f}) × UNIV"

<proof>

lemma *lift_preserves_eq*:

"(lift i F ∈ preserves v) = (F ∈ preserves (v o lift_map i))"

<proof>

lemma *lift_preserves_sub*:

"F ∈ preserves snd"

```

==> lift i F ∈ preserves (v o sub j o fst) =
      (if i=j then F ∈ preserves (v o fst) else True)"
⟨proof⟩

```

12.7 Lemmas to Handle Function Composition (o) More Consistently

```

lemma o_equiv_assoc: "f o g = h ==> f' o f o g = f' o h"
⟨proof⟩

```

```

lemma o_equiv_apply: "f o g = h ==> ∀x. f(g x) = h x"
⟨proof⟩

```

```

lemma fst_o_lift_map: "sub i o fst o lift_map i = fst"
⟨proof⟩

```

```

lemma snd_o_lift_map: "snd o lift_map i = snd o snd"
⟨proof⟩

```

12.8 More lemmas about extend and project

They could be moved to theory Extend or Project

```

lemma extend_act_extend_act:
  "extend_act h' (extend_act h act) =
   extend_act (%(x,(y,y')). h'(h(x,y),y')) act"
⟨proof⟩

```

```

lemma project_act_project_act:
  "project_act h (project_act h' act) =
   project_act (%(x,(y,y')). h'(h(x,y),y')) act"
⟨proof⟩

```

```

lemma project_act_extend_act:
  "project_act h (extend_act h' act) =
   {(x,x'). ∃s s' y y' z. (s,s') ∈ act &
    h(x,y) = h'(s,z) & h(x',y') = h'(s',z)}"
⟨proof⟩

```

12.9 OK and "lift"

```

lemma act_in_UNION_preserves_fst:
  "act ⊆ {(x,x'). fst x = fst x'} ==> act ∈ ⋃ (Acts ' (preserves fst))"
⟨proof⟩

```

```

lemma UNION_OK_lift_I:
  "[| ∀i ∈ I. F i ∈ preserves snd;
   ∀i ∈ I. ⋃ (Acts ' (preserves fst)) ⊆ AllowedActs (F i) |]
  ==> OK I (%i. lift i (F i))"
⟨proof⟩

```

```

lemma OK_lift_I:
  "[| ∀i ∈ I. F i ∈ preserves snd;
   ∀i ∈ I. preserves fst ⊆ Allowed (F i) |]

```

```

      ==> OK I (%i. lift i (F i))"
⟨proof⟩

lemma Allowed_lift [simp]: "Allowed (lift i F) = lift i ` (Allowed F)"
⟨proof⟩

lemma lift_image_preserves:
  "lift i ` preserves v = preserves (v o drop_map i)"
⟨proof⟩

end

theory PPROD imports Lift_prog begin

definition PLam :: "[nat set, nat => ('b * ((nat=>'b) * 'c)) program]
  => ((nat=>'b) * 'c) program" where
  "PLam I F ==  $\bigsqcup_{i \in I} \text{lift } i (F i)$ "

syntax
  "_PLam" :: "[pttrn, nat set, 'b set] => (nat => 'b) set" ("( $\text{3plam } \_ : \_ / \_$ )" 10)
translations
  "plam x : A. B" == "CONST PLam A (%x. B)"

lemma Init_PLam [simp]: "Init (PLam I F) = ( $\bigcap_{i \in I} \text{lift\_set } i (\text{Init } (F i))$ )"
⟨proof⟩

lemma PLam_empty [simp]: "PLam {} F = SKIP"
⟨proof⟩

lemma PLam_SKIP [simp]: "(plam i : I. SKIP) = SKIP"
⟨proof⟩

lemma PLam_insert: "PLam (insert i I) F = (lift i (F i))  $\sqcup$  (PLam I F)"
⟨proof⟩

lemma PLam_component_iff: " $((\text{PLam } I F) \leq H) = (\forall i \in I. \text{lift } i (F i) \leq H)$ "
⟨proof⟩

lemma component_PLam: " $i \in I \implies \text{lift } i (F i) \leq (\text{PLam } I F)$ "
⟨proof⟩

lemma PLam_constrains:
  "[[ i ∈ I;  $\forall j. F j \in \text{preserves snd } \text{[]}$  ]
  ==> (PLam I F ∈ (lift_set i (A × UNIV)) co
      (lift_set i (B × UNIV))) =

```

```
(F i ∈ (A × UNIV) co (B × UNIV))"
⟨proof⟩
```

```
lemma PLam_stable:
  "[| i ∈ I; ∀j. F j ∈ preserves snd |]
   ==> (PLam I F ∈ stable (lift_set i (A × UNIV))) =
        (F i ∈ stable (A × UNIV))"
⟨proof⟩
```

```
lemma PLam_transient:
  "i ∈ I ==>
   PLam I F ∈ transient A = (∃i ∈ I. lift i (F i) ∈ transient A)"
⟨proof⟩
```

This holds because the $F j$ cannot change $\text{lift_set } i$

```
lemma PLam_ensures:
  "[| i ∈ I; F i ∈ (A × UNIV) ensures (B × UNIV);
     ∀j. F j ∈ preserves snd |]
   ==> PLam I F ∈ lift_set i (A × UNIV) ensures lift_set i (B × UNIV)"
⟨proof⟩
```

```
lemma PLam_leadsTo_Basis:
  "[| i ∈ I;
     F i ∈ ((A × UNIV) - (B × UNIV)) co
            ((A × UNIV) ∪ (B × UNIV));
     F i ∈ transient ((A × UNIV) - (B × UNIV));
     ∀j. F j ∈ preserves snd |]
   ==> PLam I F ∈ lift_set i (A × UNIV) leadsTo lift_set i (B × UNIV)"
⟨proof⟩
```

```
lemma invariant_imp_PLam_invariant:
  "[| F i ∈ invariant (A × UNIV); i ∈ I;
     ∀j. F j ∈ preserves snd |]
   ==> PLam I F ∈ invariant (lift_set i (A × UNIV))"
⟨proof⟩
```

```
lemma PLam_preserves_fst [simp]:
  "∀j. F j ∈ preserves snd
   ==> (PLam I F ∈ preserves (v o sub j o fst)) =
        (if j ∈ I then F j ∈ preserves (v o fst) else True)"
⟨proof⟩
```

```
lemma PLam_preserves_snd [simp,intro]:
  "∀j. F j ∈ preserves snd ==> PLam I F ∈ preserves snd"
⟨proof⟩
```

This rule looks unsatisfactory because it refers to *lift*. One must use *lift_guarantees_eq_lift_inv* to rewrite the first subgoal and something like *lift_preserves_sub* to rewrite the third. However there's no obvious alternative for the third premise.

```

lemma guarantees_PLam_I:
  "[| lift i (F i) ∈ X guarantees Y; i ∈ I;
    OK I (λi. lift i (F i)) |]
  ==> (PLam I F) ∈ X guarantees Y"
⟨proof⟩

lemma Allowed_PLam [simp]:
  "Allowed (PLam I F) = (∩ i ∈ I. lift i ' Allowed(F i))"
⟨proof⟩

lemma PLam_preserves [simp]:
  "(PLam I F) ∈ preserves v = (∀ i ∈ I. F i ∈ preserves (v o lift_map i))"
⟨proof⟩

end

```

13 The Prefix Ordering on Lists

```

theory ListOrder
imports Main
begin

inductive_set
  genPrefix :: "('a * 'a)set => ('a list * 'a list)set"
  for r :: "('a * 'a)set"
where
  Nil:      "([], []) ∈ genPrefix(r)"

  | prepend: "[| (xs,ys) ∈ genPrefix(r); (x,y) ∈ r |] ==>
    (x#xs, y#ys) ∈ genPrefix(r)"

  | append: "(xs,ys) ∈ genPrefix(r) ==> (xs, ys@zs) ∈ genPrefix(r)"

instantiation list :: (type) ord
begin

definition
  prefix_def:      "xs <= zs ↔ (xs, zs) ∈ genPrefix Id"

definition
  strict_prefix_def: "xs < zs ↔ xs ≤ zs ∧ ¬ zs ≤ (xs :: 'a list)"

instance ⟨proof⟩

```

end

definition Le :: "(nat*nat) set" where
 "Le == {(x,y). x <= y}"

definition Ge :: "(nat*nat) set" where
 "Ge == {(x,y). y <= x}"

abbreviation

$pfixLe$:: "[nat list, nat list] => bool" (infixl "pfixLe" 50) where
 "xs pfixLe ys == (xs,ys) ∈ genPrefix Le"

abbreviation

$pfixGe$:: "[nat list, nat list] => bool" (infixl "pfixGe" 50) where
 "xs pfixGe ys == (xs,ys) ∈ genPrefix Ge"

13.1 preliminary lemmas

lemma $Nil_genPrefix$ [iff]: "([], xs) ∈ genPrefix r"
 ⟨proof⟩

lemma $genPrefix_length_le$: "(xs,ys) ∈ genPrefix r ⇒ length xs <= length ys"
 ⟨proof⟩

lemma $cdlemma$:

"[| (xs', ys') ∈ genPrefix r |]
 ==> (∀ x xs. xs' = x#xs → (∃ y ys. ys' = y#ys & (x,y) ∈ r & (xs, ys) ∈ genPrefix r))"
 ⟨proof⟩

lemma $cons_genPrefixE$ [elim!]:

"[| (x#xs, zs) ∈ genPrefix r;
 !!y ys. [| zs = y#ys; (x,y) ∈ r; (xs, ys) ∈ genPrefix r |] ==>
 P
 |] ==> P"
 ⟨proof⟩

lemma $Cons_genPrefix_Cons$ [iff]:

"((x#xs,y#ys) ∈ genPrefix r) = ((x,y) ∈ r ∧ (xs,ys) ∈ genPrefix r)"
 ⟨proof⟩

13.2 genPrefix is a partial order

lemma $refl_genPrefix$: "refl r ==> refl (genPrefix r)"
 ⟨proof⟩

lemma $genPrefix_refl$ [simp]: "refl r ⇒ (1,1) ∈ genPrefix r"
 ⟨proof⟩

lemma $genPrefix_mono$: "r<=s ==> genPrefix r <= genPrefix s"
 ⟨proof⟩

lemma *append_genPrefix*:

" $(xs @ ys, zs) \in \text{genPrefix } r \implies (xs, zs) \in \text{genPrefix } r$ "
 ⟨*proof*⟩

lemma *genPrefix_trans_0*:

assumes " $(x, y) \in \text{genPrefix } r$ "
 shows " $\bigwedge z. (y, z) \in \text{genPrefix } s \implies (x, z) \in \text{genPrefix } (r @ s)$ "
 ⟨*proof*⟩

lemma *genPrefix_trans*:

" $(x, y) \in \text{genPrefix } r \implies (y, z) \in \text{genPrefix } r \implies \text{trans } r$
 $\implies (x, z) \in \text{genPrefix } r$ "
 ⟨*proof*⟩

lemma *prefix_genPrefix_trans*:

" $[| x <= y; (y, z) \in \text{genPrefix } r |] \implies (x, z) \in \text{genPrefix } r$ "
 ⟨*proof*⟩

lemma *genPrefix_prefix_trans*:

" $[| (x, y) \in \text{genPrefix } r; y <= z |] \implies (x, z) \in \text{genPrefix } r$ "
 ⟨*proof*⟩

lemma *trans_genPrefix*: " $\text{trans } r \implies \text{trans } (\text{genPrefix } r)$ "

⟨*proof*⟩

lemma *genPrefix_antisym*:

assumes 1: " $(xs, ys) \in \text{genPrefix } r$ "
 and 2: " $\text{antisym } r$ "
 and 3: " $(ys, xs) \in \text{genPrefix } r$ "
 shows " $xs = ys$ "
 ⟨*proof*⟩

lemma *antisym_genPrefix*: " $\text{antisym } r \implies \text{antisym } (\text{genPrefix } r)$ "

⟨*proof*⟩

13.3 recursion equations

lemma *genPrefix_Nil [simp]*: " $((xs, []) \in \text{genPrefix } r) = (xs = [])$ "

⟨*proof*⟩

lemma *same_genPrefix_genPrefix [simp]*:

" $\text{refl } r \implies ((xs@ys, xs@zs) \in \text{genPrefix } r) = ((ys, zs) \in \text{genPrefix } r)$ "
 ⟨*proof*⟩

lemma *genPrefix_Cons*:


```

    "((xs, y#ys) ∈ genPrefix r) =
      (xs=[] | (∃z zs. xs=z#zs & (z,y) ∈ r & (zs,ys) ∈ genPrefix r))"
  ⟨proof⟩

lemma genPrefix_take_append:
  "[| refl r; (xs,ys) ∈ genPrefix r |]
    ==> (xs@zs, take (length xs) ys @ zs) ∈ genPrefix r"
  ⟨proof⟩

lemma genPrefix_append_both:
  "[| refl r; (xs,ys) ∈ genPrefix r; length xs = length ys |]
    ==> (xs@zs, ys @ zs) ∈ genPrefix r"
  ⟨proof⟩

lemma append_cons_eq: "xs @ y # ys = (xs @ [y]) @ ys"
  ⟨proof⟩

lemma aolemma:
  "[| (xs,ys) ∈ genPrefix r; refl r |]
    ==> length xs < length ys → (xs @ [ys ! length xs], ys) ∈ genPrefix
  r"
  ⟨proof⟩

lemma append_one_genPrefix:
  "[| (xs,ys) ∈ genPrefix r; length xs < length ys; refl r |]
    ==> (xs @ [ys ! length xs], ys) ∈ genPrefix r"
  ⟨proof⟩

lemma genPrefix_imp_nth:
  "i < length xs ⇒ (xs, ys) ∈ genPrefix r ⇒ (xs ! i, ys ! i) ∈ r"
  ⟨proof⟩

lemma nth_imp_genPrefix:
  "length xs ≤ length ys ⇒
    (∀i. i < length xs → (xs ! i, ys ! i) ∈ r) ⇒
    (xs, ys) ∈ genPrefix r"
  ⟨proof⟩

lemma genPrefix_iff_nth:
  "((xs,ys) ∈ genPrefix r) =
    (length xs ≤ length ys & (∀i. i < length xs → (xs!i, ys!i) ∈ r))"
  ⟨proof⟩

13.4 The type of lists is partially ordered

declare refl_Id [iff]
          antisym_Id [iff]
          trans_Id [iff]

```

```

lemma prefix_refl [iff]: "xs <= (xs::'a list)"
  <proof>

lemma prefix_trans: "!!xs::'a list. [| xs <= ys; ys <= zs |] ==> xs <= zs"
  <proof>

lemma prefix_antisym: "!!xs::'a list. [| xs <= ys; ys <= xs |] ==> xs = ys"
  <proof>

lemma prefix_less_le_not_le: "!!xs::'a list. (xs < zs) = (xs <= zs & ¬ zs
  ≤ xs)"
  <proof>

instance list :: (type) order
  <proof>

lemma set_mono: "xs <= ys ==> set xs <= set ys"
  <proof>

lemma Nil_prefix [iff]: "[] <= xs"
  <proof>

lemma prefix_Nil [simp]: "(xs <= []) = (xs = [])"
  <proof>

lemma Cons_prefix_Cons [simp]: "(x#xs <= y#ys) = (x=y & xs<=ys)"
  <proof>

lemma same_prefix_prefix [simp]: "(xs@ys <= xs@zs) = (ys <= zs)"
  <proof>

lemma append_prefix [iff]: "(xs@ys <= xs) = (ys <= [])"
  <proof>

lemma prefix_appendI [simp]: "xs <= ys ==> xs <= ys@zs"
  <proof>

lemma prefix_Cons:
  "(xs <= y#ys) = (xs=[] | (∃zs. xs=y#zs ∧ zs <= ys))"
  <proof>

lemma append_one_prefix:
  "[| xs <= ys; length xs < length ys |] ==> xs @ [ys ! length xs] <= ys"
  <proof>

lemma prefix_length_le: "xs <= ys ==> length xs <= length ys"
  <proof>

lemma spllemma: "xs<=ys ==> xs~=ys --> length xs < length ys"
  <proof>

```

lemma *strict_prefix_length_less*: "xs < ys ==> length xs < length ys"
 <proof>

lemma *mono_length*: "mono length"
 <proof>

lemma *prefix_iff*: "(xs <= zs) = (\exists ys. zs = xs@ys)"
 <proof>

lemma *prefix_snoc [simp]*: "(xs <= ys@[y]) = (xs = ys@[y] | xs <= ys)"
 <proof>

lemma *prefix_append_iff*:
 "(xs <= ys@zs) = (xs <= ys | (\exists us. xs = ys@us & us <= zs))"
 <proof>

lemma *common_prefix_linear*:
 fixes xs ys zs :: "'a list"
 shows "xs <= zs ==> ys <= zs ==> xs <= ys | ys <= xs"
 <proof>

13.5 *pfixLe, pfixGe: properties inherited from the translations*

lemma *refl_Le [iff]*: "refl Le"
 <proof>

lemma *antisym_Le [iff]*: "antisym Le"
 <proof>

lemma *trans_Le [iff]*: "trans Le"
 <proof>

lemma *pfixLe_refl [iff]*: "x pfixLe x"
 <proof>

lemma *pfixLe_trans*: "[| x pfixLe y; y pfixLe z |] ==> x pfixLe z"
 <proof>

lemma *pfixLe_antisym*: "[| x pfixLe y; y pfixLe x |] ==> x = y"
 <proof>

lemma *prefix_imp_pfixLe*: "xs<=ys ==> xs pfixLe ys"
 <proof>

lemma *refl_Ge [iff]*: "refl Ge"
 <proof>

lemma *antisym_Ge [iff]*: "antisym Ge"
 <proof>

```

lemma trans_Ge [iff]: "trans Ge"
  <proof>

lemma prefixGe_refl [iff]: "x prefixGe x"
  <proof>

lemma prefixGe_trans: "[| x prefixGe y; y prefixGe z |] ==> x prefixGe z"
  <proof>

lemma prefixGe_antisym: "[| x prefixGe y; y prefixGe x |] ==> x = y"
  <proof>

lemma prefix_imp_prefixGe: "xs<=ys ==> xs prefixGe ys"
  <proof>

end

```

14 The Follows Relation of Charpentier and Sivilotte

```

theory Follows
imports SubstAx ListOrder "HOL-Library.Multiset"
begin

definition Follows :: "[ 'a => 'b::{order}, 'a => 'b::{order} ] => 'a program
  set" (infixl "Fols" 65) where
  "f Fols g == Increasing g  $\cap$  Increasing f Int
    Always {s. f s  $\leq$  g s} Int
    ( $\bigcap$  k. {s. k  $\leq$  g s} LeadsTo {s. k  $\leq$  f s})"

lemma mono_Always_o:
  "mono h ==> Always {s. f s  $\leq$  g s}  $\subseteq$  Always {s. h (f s)  $\leq$  h (g s)}"
  <proof>

lemma mono_LeadsTo_o:
  "mono (h::'a::order => 'b::order)
    ==> ( $\bigcap$  j. {s. j  $\leq$  g s} LeadsTo {s. j  $\leq$  f s})  $\subseteq$ 
    ( $\bigcap$  k. {s. k  $\leq$  h (g s)} LeadsTo {s. k  $\leq$  h (f s)})"
  <proof>

lemma Follows_constant [iff]: "F  $\in$  (%s. c) Fols (%s. c)"
  <proof>

lemma mono_Follows_o:
  assumes "mono h"
  shows "f Fols g  $\subseteq$  (h o f) Fols (h o g)"
  <proof>

lemma mono_Follows_apply:
  "mono h ==> f Fols g  $\subseteq$  (%x. h (f x)) Fols (%x. h (g x))"
  <proof>

```

lemma *Follows_trans*:
 "[| F ∈ f Fols g; F ∈ g Fols h |] ==> F ∈ f Fols h"
 ⟨proof⟩

14.1 Destruction rules

lemma *Follows_Increasing1*: "F ∈ f Fols g ==> F ∈ Increasing f"
 ⟨proof⟩

lemma *Follows_Increasing2*: "F ∈ f Fols g ==> F ∈ Increasing g"
 ⟨proof⟩

lemma *Follows_Bounded*: "F ∈ f Fols g ==> F ∈ Always {s. f s ≤ g s}"
 ⟨proof⟩

lemma *Follows_LeadsTo*:
 "F ∈ f Fols g ==> F ∈ {s. k ≤ g s} LeadsTo {s. k ≤ f s}"
 ⟨proof⟩

lemma *Follows_LeadsTo_prefixLe*:
 "F ∈ f Fols g ==> F ∈ {s. k prefixLe g s} LeadsTo {s. k prefixLe f s}"
 ⟨proof⟩

lemma *Follows_LeadsTo_prefixGe*:
 "F ∈ f Fols g ==> F ∈ {s. k prefixGe g s} LeadsTo {s. k prefixGe f s}"
 ⟨proof⟩

lemma *Always_Follows1*:
 "[| F ∈ Always {s. f s = f' s}; F ∈ f Fols g |] ==> F ∈ f' Fols g"
 ⟨proof⟩

lemma *Always_Follows2*:
 "[| F ∈ Always {s. g s = g' s}; F ∈ f Fols g |] ==> F ∈ f Fols g'"
 ⟨proof⟩

14.2 Union properties (with the subset ordering)

lemma *increasing_Un*:
 "[| F ∈ increasing f; F ∈ increasing g |]
 ==> F ∈ increasing (%s. (f s) ∪ (g s))"
 ⟨proof⟩

lemma *Increasing_Un*:
 "[| F ∈ Increasing f; F ∈ Increasing g |]
 ==> F ∈ Increasing (%s. (f s) ∪ (g s))"
 ⟨proof⟩

lemma *Always_Un*:
 "[| F ∈ Always {s. f' s ≤ f s}; F ∈ Always {s. g' s ≤ g s} |]
 ==> F ∈ Always {s. f' s ∪ g' s ≤ f s ∪ g s}"
 ⟨proof⟩

```

lemma Follows_Un_lemma:
  "[| F ∈ Increasing f; F ∈ Increasing g;
    F ∈ Increasing g'; F ∈ Always {s. f' s ≤ f s};
    ∀k. F ∈ {s. k ≤ f s} LeadsTo {s. k ≤ f' s} |]"
  ==> F ∈ {s. k ≤ f s ∪ g s} LeadsTo {s. k ≤ f' s ∪ g s}"
<proof>

```

```

lemma Follows_Un:
  "[| F ∈ f' Fols f; F ∈ g' Fols g |]"
  ==> F ∈ (%s. (f' s) ∪ (g' s)) Fols (%s. (f s) ∪ (g s))"
<proof>

```

14.3 Multiset union properties (with the multiset ordering)

```

lemma increasing_union:
  "[| F ∈ increasing f; F ∈ increasing g |]"
  ==> F ∈ increasing (%s. (f s) + (g s :: ('a::order) multiset))"
<proof>

```

```

lemma Increasing_union:
  "[| F ∈ Increasing f; F ∈ Increasing g |]"
  ==> F ∈ Increasing (%s. (f s) + (g s :: ('a::order) multiset))"
<proof>

```

```

lemma Always_union:
  "[| F ∈ Always {s. f' s ≤ f s}; F ∈ Always {s. g' s ≤ g s} |]"
  ==> F ∈ Always {s. f' s + g' s ≤ f s + (g s :: ('a::order) multiset)}"
<proof>

```

```

lemma Follows_union_lemma:
  "[| F ∈ Increasing f; F ∈ Increasing g;
    F ∈ Increasing g'; F ∈ Always {s. f' s ≤ f s};
    ∀k::('a::order) multiset.
    F ∈ {s. k ≤ f s} LeadsTo {s. k ≤ f' s} |]"
  ==> F ∈ {s. k ≤ f s + g s} LeadsTo {s. k ≤ f' s + g s}"
<proof>

```

```

lemma Follows_union:
  "!!g g' ::'b => ('a::order) multiset.
  [| F ∈ f' Fols f; F ∈ g' Fols g |]"
  ==> F ∈ (%s. (f' s) + (g' s)) Fols (%s. (f s) + (g s))"
<proof>

```

```

lemma Follows_sum:
  "!!f ::['c,'b] => ('a::order) multiset.
  [| ∀i ∈ I. F ∈ f' i Fols f i; finite I |]"
  ==> F ∈ (%s. ∑ i ∈ I. f' i s) Fols (%s. ∑ i ∈ I. f i s)"
<proof>

```

lemma *Increasing_imp_Stable_prefixGe*:
 "F ∈ Increasing func ==> F ∈ Stable {s. h prefixGe (func s)}"
 ⟨proof⟩

lemma *LeadsTo_le_imp_prefixGe*:
 "∀z. F ∈ {s. z ≤ f s} LeadsTo {s. z ≤ g s}
 ==> F ∈ {s. z prefixGe f s} LeadsTo {s. z prefixGe g s}"
 ⟨proof⟩

end

15 Predicate Transformers

theory *Transformers* imports *Comp* begin

15.1 Defining the Predicate Transformers *wp*, *awp* and *wens*

definition *wp* :: "[('a*'a) set, 'a set] => 'a set" where
 — Dijkstra's weakest-precondition operator (for an individual command)
 "wp act B == - (act⁻¹ '(-B))"

definition *awp* :: "['a program, 'a set] => 'a set" where
 — Dijkstra's weakest-precondition operator (for a program)
 "awp F B == (∩ act ∈ Acts F. wp act B)"

definition *wens* :: "['a program, ('a*'a) set, 'a set] => 'a set" where
 — The weakest-ensures transformer
 "wens F act B == gfp(λX. (wp act B ∩ awp F (B ∪ X)) ∪ B)"

The fundamental theorem for wp

theorem *wp_iff*: "(A ≤ wp act B) = (act 'A ≤ B)"
 ⟨proof⟩

This lemma is a good deal more intuitive than the definition!

lemma *in_wp_iff*: "(a ∈ wp act B) = (∀x. (a,x) ∈ act --> x ∈ B)"
 ⟨proof⟩

lemma *Compl_Domain_subset_wp*: "- (Domain act) ⊆ wp act B"
 ⟨proof⟩

lemma *wp_empty [simp]*: "wp act {} = - (Domain act)"
 ⟨proof⟩

The identity relation is the skip action

lemma *wp_Id [simp]*: "wp Id B = B"
 ⟨proof⟩

lemma *wp_totalize_act*:
 "wp (totalize_act act) B = (wp act B ∩ Domain act) ∪ (B - Domain act)"
 ⟨proof⟩

lemma *awp_subset*: " $(awp\ F\ A \subseteq A)$ "
 ⟨*proof*⟩

lemma *awp_Int_eq*: " $awp\ F\ (A \cap B) = awp\ F\ A \cap awp\ F\ B$ "
 ⟨*proof*⟩

The fundamental theorem for awp

theorem *awp_iff_constrains*: " $(A \leq awp\ F\ B) = (F \in A\ co\ B)$ "
 ⟨*proof*⟩

lemma *awp_iff_stable*: " $(A \subseteq awp\ F\ A) = (F \in stable\ A)$ "
 ⟨*proof*⟩

lemma *stable_imp_awp_ident*: " $F \in stable\ A \implies awp\ F\ A = A$ "
 ⟨*proof*⟩

lemma *wp_mono*: " $(A \subseteq B) \implies wp\ act\ A \subseteq wp\ act\ B$ "
 ⟨*proof*⟩

lemma *awp_mono*: " $(A \subseteq B) \implies awp\ F\ A \subseteq awp\ F\ B$ "
 ⟨*proof*⟩

lemma *wens_unfold*:
 " $wens\ F\ act\ B = (wp\ act\ B \cap awp\ F\ (B \cup wens\ F\ act\ B)) \cup B$ "
 ⟨*proof*⟩

lemma *wens_Id [simp]*: " $wens\ F\ Id\ B = B$ "
 ⟨*proof*⟩

These two theorems justify the claim that *wens* returns the weakest assertion satisfying the ensures property

lemma *ensures_imp_wens*: " $F \in A\ ensures\ B \implies \exists act \in Acts\ F. A \subseteq wens\ F\ act\ B$ "
 ⟨*proof*⟩

lemma *wens_ensures*: " $act \in Acts\ F \implies F \in (wens\ F\ act\ B)\ ensures\ B$ "
 ⟨*proof*⟩

These two results constitute assertion (4.13) of the thesis

lemma *wens_mono*: " $(A \subseteq B) \implies wens\ F\ act\ A \subseteq wens\ F\ act\ B$ "
 ⟨*proof*⟩

lemma *wens_weakening*: " $B \subseteq wens\ F\ act\ B$ "
 ⟨*proof*⟩

Assertion (6), or 4.16 in the thesis

lemma *subset_wens*: " $A - B \subseteq wp\ act\ B \cap awp\ F\ (B \cup A) \implies A \subseteq wens\ F\ act\ B$ "
 ⟨*proof*⟩

Assertion 4.17 in the thesis

lemma *Diff_wens_constrains*: " $F \in (wens\ F\ act\ A - A)\ co\ wens\ F\ act\ A$ "

<proof>

Assertion (7): 4.18 in the thesis. NOTE that many of these results hold for an arbitrary action. We often do not require $act \in Acts F$

lemma *stable_wens*: " $F \in stable A \implies F \in stable (wens F act A)$ "
<proof>

Assertion 4.20 in the thesis.

lemma *wens_Int_eq_lemma*:
 " $[|T-B \subseteq awp F T; act \in Acts F|] \implies T \cap wens F act B \subseteq wens F act (T \cap B)$ "
<proof>

Assertion (8): 4.21 in the thesis. Here we indeed require $act \in Acts F$

lemma *wens_Int_eq*:
 " $[|T-B \subseteq awp F T; act \in Acts F|] \implies T \cap wens F act B = T \cap wens F act (T \cap B)$ "
<proof>

15.2 Defining the Weakest Ensures Set

inductive_set

wens_set :: " $['a program, 'a set] \Rightarrow 'a set set$ "
 for F :: " $'a program$ " and B :: " $'a set$ "
 where

Basis: " $B \in wens_set F B$ "

| *Wens*: " $[|X \in wens_set F B; act \in Acts F|] \implies wens F act X \in wens_set F B$ "

| *Union*: " $W \neq \{\} \implies \forall U \in W. U \in wens_set F B \implies \bigcup W \in wens_set F B$ "

lemma *wens_set_imp_co*: " $A \in wens_set F B \implies F \in (A-B) co A$ "
<proof>

lemma *wens_set_imp_leadsTo*: " $A \in wens_set F B \implies F \in A leadsTo B$ "
<proof>

lemma *leadsTo_imp_wens_set*: " $F \in A leadsTo B \implies \exists C \in wens_set F B. A \subseteq C$ "
<proof>

Assertion (9): 4.27 in the thesis.

lemma *leadsTo_iff_wens_set*: " $(F \in A leadsTo B) = (\exists C \in wens_set F B. A \subseteq C)$ "
<proof>

This is the result that requires the definition of *wens_set* to require W to be non-empty in the Unio case, for otherwise we should always have $\{\} \in wens_set F B$.

lemma *wens_set_imp_subset*: " $A \in wens_set F B \implies B \subseteq A$ "
<proof>

15.3 Properties Involving Program Union

Assertion (4.30) of thesis, reoriented

lemma *awp_Join_eq*: " $\text{awp } (F \sqcup G) B = \text{awp } F B \cap \text{awp } G B$ "
 $\langle \text{proof} \rangle$

lemma *wens_subset*: " $\text{wens } F \text{ act } B - B \subseteq \text{wp } \text{act } B \cap \text{awp } F (B \cup \text{wens } F \text{ act } B)$ "
 $\langle \text{proof} \rangle$

Assertion (4.31)

lemma *subset_wens_Join*:
 $[[A = T \cap \text{wens } F \text{ act } B; T - B \subseteq \text{awp } F T; A - B \subseteq \text{awp } G (A \cup B)]]$
 $\implies A \subseteq \text{wens } (F \sqcup G) \text{ act } B$
 $\langle \text{proof} \rangle$

Assertion (4.32)

lemma *wens_Join_subset*: " $\text{wens } (F \sqcup G) \text{ act } B \subseteq \text{wens } F \text{ act } B$ "
 $\langle \text{proof} \rangle$

Lemma, because the inductive step is just too messy.

lemma *wens_Union_inductive_step*:
assumes *awpF*: " $T - B \subseteq \text{awp } F T$ "
and *awpG*: " $!!X. X \in \text{wens_set } F B \implies (T \cap X) - B \subseteq \text{awp } G (T \cap X)$ "
shows " $[X \in \text{wens_set } F B; \text{act} \in \text{Acts } F; Y \subseteq X; T \cap X = T \cap Y]$ "
 $\implies \text{wens } (F \sqcup G) \text{ act } Y \subseteq \text{wens } F \text{ act } X \wedge$
 $T \cap \text{wens } F \text{ act } X = T \cap \text{wens } (F \sqcup G) \text{ act } Y$
 $\langle \text{proof} \rangle$

theorem *wens_Union*:
assumes *awpF*: " $T - B \subseteq \text{awp } F T$ "
and *awpG*: " $!!X. X \in \text{wens_set } F B \implies (T \cap X) - B \subseteq \text{awp } G (T \cap X)$ "
and *major*: " $X \in \text{wens_set } F B$ "
shows " $\exists Y \in \text{wens_set } (F \sqcup G) B. Y \subseteq X \ \& \ T \cap X = T \cap Y$ "
 $\langle \text{proof} \rangle$

theorem *leadsTo_Join*:
assumes *leadsTo*: " $F \in A \text{ leadsTo } B$ "
and *awpF*: " $T - B \subseteq \text{awp } F T$ "
and *awpG*: " $!!X. X \in \text{wens_set } F B \implies (T \cap X) - B \subseteq \text{awp } G (T \cap X)$ "
shows " $F \sqcup G \in T \cap A \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

15.4 The Set $\text{wens_set } F B$ for a Single-Assignment Program

Thesis Section 4.3.3

We start by proving laws about single-assignment programs

lemma *awp_single_eq* [*simp*]:
 $\text{awp } (\text{mk_program } (\text{init}, \{\text{act}\}, \text{allowed})) B = B \cap \text{wp } \text{act } B$
 $\langle \text{proof} \rangle$

lemma wp_Un_subset : " $wp\ act\ A\ \cup\ wp\ act\ B\ \subseteq\ wp\ act\ (A\ \cup\ B)$ "
 <proof>

lemma wp_Un_eq : " $single_valued\ act\ ==>\ wp\ act\ (A\ \cup\ B) = wp\ act\ A\ \cup\ wp\ act\ B$ "
 <proof>

lemma wp_UN_subset : " $(\bigcup_{i \in I}. wp\ act\ (A\ i))\ \subseteq\ wp\ act\ (\bigcup_{i \in I}. A\ i)$ "
 <proof>

lemma wp_UN_eq :
 "[$single_valued\ act; I \neq \{\}$]"
 $==>\ wp\ act\ (\bigcup_{i \in I}. A\ i) = (\bigcup_{i \in I}. wp\ act\ (A\ i))$
 <proof>

lemma $wens_single_eq$:
 " $wens\ (mk_program\ (init,\ \{act\},\ allowed))\ act\ B = B\ \cup\ wp\ act\ B$ "
 <proof>

Next, we express the $wens_set$ for single-assignment programs

definition $wens_single_finite$:: " $(\ 'a * 'a) set, 'a set, nat] => 'a set$ " where

$wens_single_finite\ act\ B\ k == \bigcup_{i \in atMost\ k}. (wp\ act\ \hat{\wedge}\ i)\ B$

definition $wens_single$:: " $(\ 'a * 'a) set, 'a set] => 'a set$ " where
 " $wens_single\ act\ B == \bigcup_{i}. (wp\ act\ \hat{\wedge}\ i)\ B$ "

lemma $wens_single_Un_eq$:
 " $single_valued\ act$ "
 $==>\ wens_single\ act\ B\ \cup\ wp\ act\ (wens_single\ act\ B) = wens_single\ act\ B$
 <proof>

lemma $atMost_nat_nonempty$: " $atMost\ (k::nat) \neq \{\}$ "
 <proof>

lemma $wens_single_finite_0$ [$simp$]: " $wens_single_finite\ act\ B\ 0 = B$ "
 <proof>

lemma $wens_single_finite_Suc$:
 " $single_valued\ act$ "
 $==>\ wens_single_finite\ act\ B\ (Suc\ k) =$
 $wens_single_finite\ act\ B\ k\ \cup\ wp\ act\ (wens_single_finite\ act\ B\ k)$
 <proof>

lemma $wens_single_finite_Suc_eq_wens$:
 " $single_valued\ act$ "
 $==>\ wens_single_finite\ act\ B\ (Suc\ k) =$
 $wens\ (mk_program\ (init,\ \{act\},\ allowed))\ act$
 $(wens_single_finite\ act\ B\ k)$
 <proof>

lemma $def_wens_single_finite_Suc_eq_wens$:
 "[$F = mk_program\ (init,\ \{act\},\ allowed); single_valued\ act$]"

```

    ==> wens_single_finite act B (Suc k) =
          wens F act (wens_single_finite act B k)"
  <proof>

lemma wens_single_finite_Un_eq:
  "single_valued act
   ==> wens_single_finite act B k  $\cup$  wp act (wens_single_finite act B k)
         $\in$  range (wens_single_finite act B)"
  <proof>

lemma wens_single_eq_Union:
  "wens_single act B =  $\bigcup$  (range (wens_single_finite act B))"
  <proof>

lemma wens_single_finite_eq_Union:
  "wens_single_finite act B n = ( $\bigcup_{k \in \text{atMost } n} \text{wens\_single\_finite act B } k$ )"
  <proof>

lemma wens_single_finite_mono:
  " $m \leq n \implies \text{wens\_single\_finite act B } m \subseteq \text{wens\_single\_finite act B } n$ "
  <proof>

lemma wens_single_finite_subset_wens_single:
  "wens_single_finite act B k  $\subseteq$  wens_single act B"
  <proof>

lemma subset_wens_single_finite:
  "[|W  $\subseteq$  wens_single_finite act B ‘ (atMost k); single_valued act; W  $\neq$  {}|]
   ==>  $\exists m. \bigcup W = \text{wens\_single\_finite act B } m$ "
  <proof>

lemma for Union case

lemma Union_eq_wens_single:
  "[| $\forall k. \neg W \subseteq \text{wens\_single\_finite act B ‘ } \{..k\};$ 
   W  $\subseteq$  insert (wens_single act B)
             (range (wens_single_finite act B))|]
   ==>  $\bigcup W = \text{wens\_single act B}$ "
  <proof>

lemma wens_set_subset_single:
  "single_valued act
   ==> wens_set (mk_program (init, {act}, allowed)) B  $\subseteq$ 
        insert (wens_single act B) (range (wens_single_finite act B))"
  <proof>

lemma wens_single_finite_in_wens_set:
  "single_valued act  $\implies$ 
   wens_single_finite act B k
    $\in$  wens_set (mk_program (init, {act}, allowed)) B"
  <proof>

lemma single_subset_wens_set:
  "single_valued act

```

```

==> insert (wens_single act B) (range (wens_single_finite act B)) ⊆
      wens_set (mk_program (init, {act}, allowed)) B"
⟨proof⟩

```

Theorem (4.29)

```

theorem wens_set_single_eq:
  "[[F = mk_program (init, {act}, allowed); single_valued act]]
  ==> wens_set F B =
      insert (wens_single act B) (range (wens_single_finite act B))"
⟨proof⟩

```

Generalizing Misra's Fixed Point Union Theorem (4.41)

```

lemma fp_leadsTo_Join:
  "[[T-B ⊆ awp F T; T-B ⊆ FP G; F ∈ A leadsTo B]] ==> F ⊔ G ∈ T ∩ A leadsTo
  B"
⟨proof⟩

```

end

16 Progress Sets

theory ProgressSets imports Transformers begin

16.1 Complete Lattices and the Operator *cl*

```

definition lattice :: "'a set set => bool" where
  — Meier calls them closure sets, but they are just complete lattices
  "lattice L ==
    (∀M. M ⊆ L --> ⋂M ∈ L) & (∀M. M ⊆ L --> ⋃M ∈ L)"

```

```

definition cl :: "[ 'a set set, 'a set ] => 'a set" where
  — short for “closure”
  "cl L r == ⋂{x. x ∈ L & r ⊆ x}"

```

```

lemma UNIV_in_lattice: "lattice L ==> UNIV ∈ L"
⟨proof⟩

```

```

lemma empty_in_lattice: "lattice L ==> {} ∈ L"
⟨proof⟩

```

```

lemma Union_in_lattice: "[[M ⊆ L; lattice L]] ==> ⋃M ∈ L"
⟨proof⟩

```

```

lemma Inter_in_lattice: "[[M ⊆ L; lattice L]] ==> ⋂M ∈ L"
⟨proof⟩

```

```

lemma UN_in_lattice:
  "[[lattice L; !!i. i ∈ I ==> r i ∈ L]] ==> (⋃i ∈ I. r i) ∈ L"
⟨proof⟩

```

```

lemma INT_in_lattice:
  "[[lattice L; !!i. i ∈ I ==> r i ∈ L]] ==> (⋂i ∈ I. r i) ∈ L"

```

<proof>

lemma *Un_in_lattice*: "[$x \in L; y \in L; \text{lattice } L$] $\implies x \cup y \in L$ "

<proof>

lemma *Int_in_lattice*: "[$x \in L; y \in L; \text{lattice } L$] $\implies x \cap y \in L$ "

<proof>

lemma *lattice_stable*: " $\text{lattice } \{X. F \in \text{stable } X\}$ "

<proof>

The next three results state that $\text{cl } L \ r$ is the minimal element of L that includes r .

lemma *cl_in_lattice*: " $\text{lattice } L \implies \text{cl } L \ r \in L$ "

<proof>

lemma *cl_least*: "[$c \in L; r \subseteq c$] $\implies \text{cl } L \ r \subseteq c$ "

<proof>

The next three lemmas constitute assertion (4.61)

lemma *cl_mono*: " $r \subseteq r' \implies \text{cl } L \ r \subseteq \text{cl } L \ r'$ "

<proof>

lemma *subset_cl*: " $r \subseteq \text{cl } L \ r$ "

<proof>

A reformulation of $?r \subseteq \text{cl } ?L \ ?r$

lemma *clI*: " $x \in r \implies x \in \text{cl } L \ r$ "

<proof>

A reformulation of $[?c \in ?L; ?r \subseteq ?c] \implies \text{cl } ?L \ ?r \subseteq ?c$

lemma *clD*: "[$c \in \text{cl } L \ r; B \in L; r \subseteq B$] $\implies c \in B$ "

<proof>

lemma *cl_UN_subset*: " $(\bigcup_{i \in I}. \text{cl } L \ (r \ i)) \subseteq \text{cl } L \ (\bigcup_{i \in I}. r \ i)$ "

<proof>

lemma *cl_Un*: " $\text{lattice } L \implies \text{cl } L \ (r \cup s) = \text{cl } L \ r \cup \text{cl } L \ s$ "

<proof>

lemma *cl_UN*: " $\text{lattice } L \implies \text{cl } L \ (\bigcup_{i \in I}. r \ i) = (\bigcup_{i \in I}. \text{cl } L \ (r \ i))$ "

<proof>

lemma *cl_Int_subset*: " $\text{cl } L \ (r \cap s) \subseteq \text{cl } L \ r \cap \text{cl } L \ s$ "

<proof>

lemma *cl_idem [simp]*: " $\text{cl } L \ (\text{cl } L \ r) = \text{cl } L \ r$ "

<proof>

lemma *cl_ident*: " $r \in L \implies \text{cl } L \ r = r$ "

<proof>

lemma *cl_empty [simp]*: " $\text{lattice } L \implies \text{cl } L \ \{\} = \{\}$ "

<proof>

lemma *cl_UNIV [simp]: "lattice L ==> cl L UNIV = UNIV"*

<proof>

Assertion (4.62)

lemma *cl_ident_iff: "lattice L ==> (cl L r = r) = (r ∈ L)"*

<proof>

lemma *cl_subset_in_lattice: "[|cl L r ⊆ r; lattice L|] ==> r ∈ L"*

<proof>

16.2 Progress Sets and the Main Lemma

A progress set satisfies certain closure conditions and is a simple way of including the set *wens_set F B*.

definition *closed :: "['a program, 'a set, 'a set, 'a set set] => bool" where*

*"closed F T B L == ∀ M. ∀ act ∈ Acts F. B ⊆ M & T ∩ M ∈ L -->
T ∩ (B ∪ wp act M) ∈ L"*

definition *progress_set :: "['a program, 'a set, 'a set] => 'a set set set"*

where

*"progress_set F T B ==
{L. lattice L & B ∈ L & T ∈ L & closed F T B L}"*

lemma *closedD:*

*"[|closed F T B L; act ∈ Acts F; B ⊆ M; T ∩ M ∈ L|]
==> T ∩ (B ∪ wp act M) ∈ L"*

<proof>

Note: the formalization below replaces Meier's *q* by *B* and *m* by *X*.

Part of the proof of the claim at the bottom of page 97. It's proved separately because the argument requires a generalization over all *act ∈ Acts F*.

lemma *lattice_awp_lemma:*

assumes *TXC: "T ∩ X ∈ C"* — induction hypothesis in theorem below

and *BsubX: "B ⊆ X"* — holds in inductive step

and *latt: "lattice C"*

and *TC: "T ∈ C"*

and *BC: "B ∈ C"*

and *clos: "closed F T B C"*

shows *"T ∩ (B ∪ awp F (X ∪ cl C (T ∩ X))) ∈ C"*

<proof>

Remainder of the proof of the claim at the bottom of page 97.

lemma *lattice_lemma:*

assumes *TXC: "T ∩ X ∈ C"* — induction hypothesis in theorem below

and *BsubX: "B ⊆ X"* — holds in inductive step

and *act: "act ∈ Acts F"*

and *latt: "lattice C"*

and *TC: "T ∈ C"*

and *BC: "B ∈ C"*

```

    and clos: "closed F T B C"
    shows "T  $\cap$  (wp act X  $\cap$  awp F (X  $\cup$  cl C (T $\cap$ r)))  $\cup$  X)  $\in$  C"
  <proof>

```

Induction step for the main lemma

```

lemma progress_induction_step:
  assumes TXC: "T $\cap$ X  $\in$  C" — induction hypothesis in theorem below
  and act: "act  $\in$  Acts F"
  and Xwens: "X  $\in$  wens_set F B"
  and latt: "lattice C"
  and TC: "T  $\in$  C"
  and BC: "B  $\in$  C"
  and clos: "closed F T B C"
  and Fstable: "F  $\in$  stable T"
  shows "T  $\cap$  wens F act X  $\in$  C"
  <proof>

```

Proved on page 96 of Meier's thesis. The special case when $T = UNIV$ states that every progress set for the program F and set B includes the set $wens_set F B$.

```

lemma progress_set_lemma:
  "[| C  $\in$  progress_set F T B; r  $\in$  wens_set F B; F  $\in$  stable T |] ==> T $\cap$ r
 $\in$  C"
  <proof>

```

16.3 The Progress Set Union Theorem

```

lemma closed_mono:
  assumes BB': "B  $\subseteq$  B'"
  and TBwp: "T  $\cap$  (B  $\cup$  wp act M)  $\in$  C"
  and B'C: "B'  $\in$  C"
  and TC: "T  $\in$  C"
  and latt: "lattice C"
  shows "T  $\cap$  (B'  $\cup$  wp act M)  $\in$  C"
  <proof>

```

```

lemma progress_set_mono:
  assumes BB': "B  $\subseteq$  B'"
  shows
    "[| B'  $\in$  C; C  $\in$  progress_set F T B |]
    ==> C  $\in$  progress_set F T B'"
  <proof>

```

```

theorem progress_set_Union:
  assumes leadsTo: "F  $\in$  A leadsTo B'"
  and prog: "C  $\in$  progress_set F T B"
  and Fstable: "F  $\in$  stable T"
  and BB': "B  $\subseteq$  B'"
  and B'C: "B'  $\in$  C"
  and Gco: "!!X. X $\in$ C ==> G  $\in$  X-B co X"
  shows "F $\sqcup$ G  $\in$  T $\cap$ A leadsTo B'"
  <proof>

```


16.4 Some Progress Sets

lemma *UNIV_in_progress_set*: "UNIV \in progress_set F T B"
 <proof>

16.4.1 Lattices and Relations

From Meier's thesis, section 4.5.3

definition *relcl* :: "'a set set \Rightarrow ('a * 'a) set" **where**
 — Derived relation from a lattice
 "relcl L == {(x,y). y \in cl L {x}}"

definition *latticeof* :: "('a * 'a) set \Rightarrow 'a set set" **where**
 — Derived lattice from a relation: the set of upwards-closed sets
 "latticeof r == {X. $\forall s t. s \in X \ \& \ (s,t) \in r \ \rightarrow t \in X}$ "

lemma *relcl_refl*: "(a,a) \in relcl L"
 <proof>

lemma *relcl_trans*:
 "[[(a,b) \in relcl L; (b,c) \in relcl L; lattice L]] \Rightarrow (a,c) \in relcl L"
 <proof>

lemma *refl_relcl*: "lattice L \Rightarrow refl (relcl L)"
 <proof>

lemma *trans_relcl*: "lattice L \Rightarrow trans (relcl L)"
 <proof>

lemma *lattice_latticeof*: "lattice (latticeof r)"
 <proof>

lemma *lattice_singletonI*:
 "[[lattice L; $\forall s. s \in X \Rightarrow \{s\} \in L$]] $\Rightarrow X \in L$ "
 <proof>

Equation (4.71) of Meier's thesis. He gives no proof.

lemma *cl_latticeof*:
 "[[refl r; trans r]
 \Rightarrow cl (latticeof r) X = {t. $\exists s. s \in X \ \& \ (s,t) \in r$]"
 <proof>

Related to (4.71).

lemma *cl_eq_Collect_relcl*:
 "lattice L \Rightarrow cl L X = {t. $\exists s. s \in X \ \& \ (s,t) \in relcl L$ "
 <proof>

Meier's theorem of section 4.5.3

theorem *latticeof_relcl_eq*: "lattice L \Rightarrow latticeof (relcl L) = L"
 <proof>

```

theorem relcl_latticeof_eq:
  "[/refl r; trans r/] ==> relcl (latticeof r) = r"
<proof>

```

16.4.2 Decoupling Theorems

```

definition decoupled :: "[ 'a program, 'a program ] => bool" where
  "decoupled F G ==
     $\forall \text{act} \in \text{Acts } F. \forall B. G \in \text{stable } B \rightarrow G \in \text{stable } (\text{wp act } B) "$ 

```

Rao's Decoupling Theorem

```

lemma stableco: "F  $\in$  stable A ==> F  $\in$  A-B co A"
<proof>

```

```

theorem decoupling:
  assumes leadsTo: "F  $\in$  A leadsTo B"
  and Gstable: "G  $\in$  stable B"
  and dec:      "decoupled F G"
  shows "F  $\sqcup$  G  $\in$  A leadsTo B"
<proof>

```

Rao's Weak Decoupling Theorem

```

theorem weak_decoupling:
  assumes leadsTo: "F  $\in$  A leadsTo B"
  and stable: "F  $\sqcup$  G  $\in$  stable B"
  and dec:    "decoupled F (F  $\sqcup$  G)"
  shows "F  $\sqcup$  G  $\in$  A leadsTo B"
<proof>

```

The "Decoupling via G' Union Theorem"

```

theorem decoupling_via_aux:
  assumes leadsTo: "F  $\in$  A leadsTo B"
  and prog: "{X. G'  $\in$  stable X}  $\in$  progress_set F UNIV B"
  and GG': "G  $\leq$  G'"
  — Beware! This is the converse of the refinement relation!
  shows "F  $\sqcup$  G  $\in$  A leadsTo B"
<proof>

```

16.5 Composition Theorems Based on Monotonicity and Commutativity

16.5.1 Commutativity of cl L and assignment.

```

definition commutes :: "[ 'a program, 'a set, 'a set, 'a set set ] => bool"
where
  "commutes F T B L ==
     $\forall M. \forall \text{act} \in \text{Acts } F. B \subseteq M \rightarrow$ 
     $\text{cl } L (T \cap \text{wp act } M) \subseteq T \cap (B \cup \text{wp act } (\text{cl } L (T \cap M))) "$ 

```

From Meier's thesis, section 4.5.6

```

lemma commutativity1_lemma:
  assumes commutes: "commutes F T B L"
  and lattice: "lattice L"

```

```

    and BL: "B ∈ L"
    and TL: "T ∈ L"
    shows "closed F T B L"
  <proof>

```

Version packaged with $\llbracket ?F \in ?A \mapsto ?B'; ?C \in \text{progress_set } ?F ?T ?B; ?F \in \text{UNITY.stable } ?T; ?B \subseteq ?B'; ?B' \in ?C; \bigwedge X. X \in ?C \implies ?G \in X - ?B \text{ co } X \rrbracket \implies ?F \sqcup ?G \in ?T \cap ?A \mapsto ?B'$

```

lemma commutativity1:
  assumes leadsTo: "F ∈ A leadsTo B"
    and lattice: "lattice L"
    and BL: "B ∈ L"
    and TL: "T ∈ L"
    and Fstable: "F ∈ stable T"
    and Gco: "!!X. X ∈ L ==> G ∈ X-B co X"
    and commutes: "commutes F T B L"
  shows "F ∪ G ∈ T ∩ A leadsTo B"
  <proof>

```

Possibly move to Relation.thy, after `single_valued`

```

definition funof :: "[('a*'b)set, 'a] => 'b" where
  "funof r == (λx. THE y. (x,y) ∈ r)"

```

```

lemma funof_eq: "[|single_valued r; (x,y) ∈ r|] ==> funof r x = y"
  <proof>

```

```

lemma funof_Pair_in:
  "[|single_valued r; x ∈ Domain r|] ==> (x, funof r x) ∈ r"
  <proof>

```

```

lemma funof_in:
  "[|r '{x} ⊆ A; single_valued r; x ∈ Domain r|] ==> funof r x ∈ A"
  <proof>

```

```

lemma funof_imp_wp: "[|funof act t ∈ A; single_valued act|] ==> t ∈ wp act A"
  <proof>

```

16.5.2 Commutativity of Functions and Relation

Thesis, page 109

From Meier's thesis, section 4.5.6

```

lemma commutativity2_lemma:
  assumes dcommutes:
    "∧ act s t. act ∈ Acts F ==> s ∈ T ==> (s, t) ∈ relcl L ==>
      s ∈ B | t ∈ B | (funof act s, funof act t) ∈ relcl L"
    and determ: "!!act. act ∈ Acts F ==> single_valued act"
    and total: "!!act. act ∈ Acts F ==> Domain act = UNIV"
    and lattice: "lattice L"
    and BL: "B ∈ L"
    and TL: "T ∈ L"
    and Fstable: "F ∈ stable T"

```

```

shows "commutes F T B L"
<proof>

```

Version packaged with $\llbracket ?F \in ?A \mapsto ?B'; ?C \in \text{progress_set } ?F ?T ?B; ?F \in \text{UNITY.stable } ?T; ?B \subseteq ?B'; ?B' \in ?C; \bigwedge X. X \in ?C \implies ?G \in X - ?B \text{ co } X \rrbracket \implies ?F \sqcup ?G \in ?T \cap ?A \mapsto ?B'$

```

lemma commutativity2:

```

```

  assumes leadsTo: "F ∈ A leadsTo B"

```

```

  and dcommutes:

```

```

    "∀act ∈ Acts F.

```

```

    ∀s ∈ T. ∀t. (s,t) ∈ relcl L -->

```

```

      s ∈ B | t ∈ B | (funof act s, funof act t) ∈ relcl

```

```

L"

```

```

  and determ: "!!act. act ∈ Acts F ==> single_valued act"

```

```

  and total: "!!act. act ∈ Acts F ==> Domain act = UNIV"

```

```

  and lattice: "lattice L"

```

```

  and BL: "B ∈ L"

```

```

  and TL: "T ∈ L"

```

```

  and Fstable: "F ∈ stable T"

```

```

  and Gco: "!!X. X ∈ L ==> G ∈ X-B co X"

```

```

  shows "F ∪ G ∈ T ∩ A leadsTo B"

```

```

<proof>

```

16.6 Monotonicity

From Meier's thesis, section 4.5.7, page 110

end

17 Comprehensive UNITY Theory

```

theory UNITY_Main

```

```

imports Detects PPROD Follows ProgressSets

```

```

begin

```

```

<ML>

```

```

end

```

```

theory Deadlock imports "../UNITY" begin

```

```

lemma "[| F ∈ (A ∩ B) co A; F ∈ (B ∩ A) co B |] ==> F ∈ stable (A ∩ B)"
<proof>

```

```

lemma Collect_le_Int_equals:

```

```

  "(⋂ i ∈ atMost n. A (Suc i) ∩ A i) = (⋂ i ∈ atMost (Suc n). A i)"

```

```

<proof>

```

```

lemma UN_Int_Compl_subset:
  "( $\bigcup i \in \text{lessThan } n. A i$ )  $\cap$  ( $\neg A n$ )  $\subseteq$ 
   ( $\bigcup i \in \text{lessThan } n. (A i) \cap (\neg A (\text{Suc } i))$ )"
  <proof>

lemma INT_Un_Compl_subset:
  "( $\bigcap i \in \text{lessThan } n. \neg A i \cup A (\text{Suc } i)$ )  $\subseteq$ 
   ( $\bigcap i \in \text{lessThan } n. \neg A i$ )  $\cup A n$ "
  <proof>

lemma INT_le_equals_Int_lemma:
  " $A 0 \cap (\neg(A n) \cap (\bigcap i \in \text{lessThan } n. \neg A i \cup A (\text{Suc } i))) = \{\}$ "
  <proof>

lemma INT_le_equals_Int:
  "( $\bigcap i \in \text{atMost } n. A i$ ) =
    $A 0 \cap (\bigcap i \in \text{lessThan } n. \neg A i \cup A(\text{Suc } i))$ "
  <proof>

lemma INT_le_Suc_equals_Int:
  "( $\bigcap i \in \text{atMost } (\text{Suc } n). A i$ ) =
    $A 0 \cap (\bigcap i \in \text{atMost } n. \neg A i \cup A(\text{Suc } i))$ "
  <proof>

lemma
  assumes zeroprem: " $F \in (A 0 \cap A (\text{Suc } n)) \text{ co } (A 0)$ "
  and allprem:
    " $\forall i. i \in \text{atMost } n \implies F \in (A(\text{Suc } i) \cap A i) \text{ co } (\neg A i \cup A(\text{Suc } i))$ "
  shows " $F \in \text{stable } (\bigcap i \in \text{atMost } (\text{Suc } n). A i)$ "
  <proof>

end

theory Common
imports "../UNITY_Main"
begin

consts
  ftime :: "nat=>nat"
  gtime :: "nat=>nat"

axiomatization where
  fmono: " $m \leq n \implies \text{ftime } m \leq \text{ftime } n$ " and
  gmono: " $m \leq n \implies \text{gtime } m \leq \text{gtime } n$ " and

```

fasc: "m ≤ ftime n" and
gasc: "m ≤ gtime n"

definition *common* :: "nat set" where
 "common == {n. ftime n = n & gtime n = n}"

definition *maxfg* :: "nat => nat set" where
 "maxfg m == {t. t ≤ max (ftime m) (gtime m)}"

lemma *common_stable*:
 "[| ∀m. F ∈ {m} Co (maxfg m); n ∈ common |]
 ==> F ∈ Stable (atMost n)"
 <proof>

lemma *common_safety*:
 "[| Init F ⊆ atMost n;
 ∀m. F ∈ {m} Co (maxfg m); n ∈ common |]
 ==> F ∈ Always (atMost n)"
 <proof>

lemma "SKIP ∈ {m} co (maxfg m)"
 <proof>

lemma "mk_total_program
 (UNIV, {range(%t.(t,ftime t)), range(%t.(t,gtime t))}, UNIV)
 ∈ {m} co (maxfg m)"
 <proof>

lemma "mk_total_program (UNIV, {range(%t.(t, max (ftime t) (gtime t)))},
 UNIV)
 ∈ {m} co (maxfg m)"
 <proof>

lemma "mk_total_program
 (UNIV, { {t, Suc t} | t. t < max (ftime t) (gtime t) }, UNIV)
 ∈ {m} co (maxfg m)"
 <proof>

lemma *leadsTo_common_lemma*:
 assumes "∀m. F ∈ {m} Co (maxfg m)"

```

    and "∀m ∈ lessThan n. F ∈ {m} LeadsTo (greaterThan m)"
    and "n ∈ common"
  shows "F ∈ (atMost n) LeadsTo common"
<proof>

```

```

lemma leadsTo_common:
  "[| ∀m. F ∈ {m} Co (maxfg m);
    ∀m ∈ -common. F ∈ {m} LeadsTo (greaterThan m);
    n ∈ common |]
  ==> F ∈ (atMost (LEAST n. n ∈ common)) LeadsTo common"
<proof>

```

```
end
```

```
theory Network imports "../UNITY" begin
```

```
datatype pvar = Sent | Rcvd | Idle
```

```
datatype pname = Aproc | Bproc
```

```
type_synonym state = "pname * pvar => nat"
```

```
locale F_props =
```

```
  fixes F
```

```
  assumes rsA: "F ∈ stable {s. s(Bproc,Rcvd) ≤ s(Aproc,Sent)}"
```

```
    and rsB: "F ∈ stable {s. s(Aproc,Rcvd) ≤ s(Bproc,Sent)}"
```

```
    and sent_nondec: "F ∈ stable {s. m ≤ s(proc,Sent)}"
```

```
    and rcvd_nondec: "F ∈ stable {s. n ≤ s(proc,Rcvd)}"
```

```
    and rcvd_idle: "F ∈ {s. s(proc,Idle) = Suc 0 & s(proc,Rcvd) = m}
      co {s. s(proc,Rcvd) = m --> s(proc,Idle) = Suc 0}"
```

```
    and sent_idle: "F ∈ {s. s(proc,Idle) = Suc 0 & s(proc,Sent) = n}
      co {s. s(proc,Sent) = n}"
```

```
begin
```

```
lemmas sent_nondec_A = sent_nondec [of _ Aproc]
```

```
  and sent_nondec_B = sent_nondec [of _ Bproc]
```

```
  and rcvd_nondec_A = rcvd_nondec [of _ Aproc]
```

```
  and rcvd_nondec_B = rcvd_nondec [of _ Bproc]
```

```
  and rcvd_idle_A = rcvd_idle [of Aproc]
```

```
  and rcvd_idle_B = rcvd_idle [of Bproc]
```

```
  and sent_idle_A = sent_idle [of Aproc]
```

```
  and sent_idle_B = sent_idle [of Bproc]
```

```
  and rs_AB = stable_Int [OF rsA rsB]
```

```
lemmas sent_nondec_AB = stable_Int [OF sent_nondec_A sent_nondec_B]
```

```
  and rcvd_nondec_AB = stable_Int [OF rcvd_nondec_A rcvd_nondec_B]
```

```
  and rcvd_idle_AB = constrains_Int [OF rcvd_idle_A rcvd_idle_B]
```

```
  and sent_idle_AB = constrains_Int [OF sent_idle_A sent_idle_B]
```

```

lemmas nondec_AB = stable_Int [OF sent_nondec_AB rcvd_nondec_AB]
  and idle_AB = constrains_Int [OF rcvd_idle_AB sent_idle_AB]

lemmas nondec_idle = constrains_Int [OF nondec_AB [unfolded stable_def] idle_AB]

lemma
  shows "F ∈ stable {s. s(Aproc,Idle) = Suc 0 & s(Bproc,Idle) = Suc 0 &
    s(Aproc,Sent) = s(Bproc,Rcvd) &
    s(Bproc,Sent) = s(Aproc,Rcvd) &
    s(Aproc,Rcvd) = m & s(Bproc,Rcvd) = n}"

  ⟨proof⟩

end

end

```

18 The Token Ring

```

theory Token
imports "../WFair"

```

```

begin

```

From Misra, "A Logic for Concurrent Programming" (1994), sections 5.2 and 13.2.

18.1 Definitions

```

datatype pstate = Hungry | Eating | Thinking
  — process states

```

```

record state =
  token :: "nat"
  proc  :: "nat => pstate"

```

```

definition HasTok :: "nat => state set" where
  "HasTok i == {s. token s = i}"

```

```

definition H :: "nat => state set" where
  "H i == {s. proc s i = Hungry}"

```

```

definition E :: "nat => state set" where
  "E i == {s. proc s i = Eating}"

```

```

definition T :: "nat => state set" where
  "T i == {s. proc s i = Thinking}"

```

```

locale Token =
  fixes N and F and nodeOrder and "next"
  defines nodeOrder_def:
    "nodeOrder j == measure(%i. ((j+N)-i) mod N) ∩ {..

```



```

and next_def:
  "next i == (Suc i) mod N"
assumes N_positive [iff]: "0 < N"
and TR2: "F ∈ (T i) co (T i ∪ H i)"
and TR3: "F ∈ (H i) co (H i ∪ E i)"
and TR4: "F ∈ (H i - HasTok i) co (H i)"
and TR5: "F ∈ (HasTok i) co (HasTok i ∪ -(E i))"
and TR6: "F ∈ (H i ∩ HasTok i) leadsTo (E i)"
and TR7: "F ∈ (HasTok i) leadsTo (HasTok (next i))"

lemma HasTok_partition: "[| s ∈ HasTok i; s ∈ HasTok j |] ==> i=j"
  <proof>

lemma not_E_eq: "(s ∉ E i) = (s ∈ H i | s ∈ T i)"
  <proof>

context Token
begin

lemma token_stable: "F ∈ stable (-(E i) ∪ (HasTok i))"
  <proof>

```

18.2 Progress under Weak Fairness

```

lemma wf_nodeOrder: "wf(nodeOrder j)"
  <proof>

lemma nodeOrder_eq:
  "[| i < N; j < N |] ==> ((next i, i) ∈ nodeOrder j) = (i ≠ j)"
  <proof>

```

From "A Logic for Concurrent Programming", but not used in Chapter 4. Note the use of *cases*. Reasoning about *leadsTo* takes practice!

```

lemma TR7_nodeOrder:
  "[| i < N; j < N |] ==>
  F ∈ (HasTok i) leadsTo ({s. (token s, i) ∈ nodeOrder j} ∪ HasTok j)"
  <proof>

```

Chapter 4 variant, the one actually used below.

```

lemma TR7_aux: "[| i < N; j < N; i ≠ j |]
  ==> F ∈ (HasTok i) leadsTo {s. (token s, i) ∈ nodeOrder j}"
  <proof>

```

```

lemma token_lemma:
  "({s. token s < N} ∩ token -' {m}) = (if m < N then token -' {m} else {})"
  <proof>

```

Misra's TR9: the token reaches an arbitrary node

```

lemma leadsTo_j: "j < N ==> F ∈ {s. token s < N} leadsTo (HasTok j)"
  <proof>

```

Misra's TR8: a hungry process eventually eats

```

lemma token_progress:
  "j < N ==> F ∈ ({s. token s < N} ∩ H j) leadsTo (E j)"
  <proof>

end

end

theory Channel imports "../UNITY_Main" begin

type_synonym state = "nat set"

consts
  F :: "state program"

definition minSet :: "nat set => nat option" where
  "minSet A == if A = {} then None else Some (LEAST x. x ∈ A)"

axiomatization where

  UC1: "F ∈ (minSet -' {Some x}) co (minSet -' (Some 'atLeast x))" and

  UC2: "F ∈ (minSet -' {Some x}) leadsTo {s. x ∉ s}"

lemma minSet_eq_SomeD: "minSet A = Some x ==> x ∈ A"
  <proof>

lemma minSet_empty [simp]: "minSet {} = None"
  <proof>

lemma minSet_nonempty: "x ∈ A ==> minSet A = Some (LEAST x. x ∈ A)"
  <proof>

lemma minSet_greaterThan:
  "F ∈ (minSet -' {Some x}) leadsTo (minSet -' (Some 'greaterThan x))"
  <proof>

lemma Channel_progress_lemma:
  "F ∈ (UNIV - {{}}) leadsTo (minSet -' (Some 'atLeast y))"
  <proof>

lemma Channel_progress: "!!y::nat. F ∈ (UNIV - {{}}) leadsTo {s. y ∉ s}"
  <proof>

end

```

```

theory Lift
imports "../UNITY_Main"
begin

record state =
  floor :: "int"           — current position of the lift
  "open" :: "bool"        — whether the door is opened at floor
  stop  :: "bool"        — whether the lift is stopped at floor
  req   :: "int set"      — for each floor, whether the lift is requested
  up    :: "bool"        — current direction of movement
  move  :: "bool"        — whether moving takes precedence over opening

axiomatization
  Min :: "int" and — least and greatest floors
  Max :: "int" — least and greatest floors
where
  Min_le_Max [iff]: "Min ≤ Max"

  — Abbreviations: the "always" part

definition
  above :: "state set"
  where "above = {s. ∃i. floor s < i & i ≤ Max & i ∈ req s}"

definition
  below :: "state set"
  where "below = {s. ∃i. Min ≤ i & i < floor s & i ∈ req s}"

definition
  queueing :: "state set"
  where "queueing = above ∪ below"

definition
  goingup :: "state set"
  where "goingup = above ∩ ({s. up s} ∪ -below)"

definition
  goingdown :: "state set"
  where "goingdown = below ∩ ({s. ~ up s} ∪ -above)"

definition
  ready :: "state set"
  where "ready = {s. stop s & ~ open s & move s}"

  — Further abbreviations

definition
  moving :: "state set"
  where "moving = {s. ~ stop s & ~ open s}"

definition
  stopped :: "state set"
  where "stopped = {s. stop s & ~ open s & ~ move s}"

```

definition

```
opened :: "state set"
where "opened = {s. stop s & open s & move s}"
```

definition

```
closed :: "state set" — but this is the same as ready!!
where "closed = {s. stop s & ~ open s & move s}"
```

definition

```
atFloor :: "int => state set"
where "atFloor n = {s. floor s = n}"
```

definition

```
Req :: "int => state set"
where "Req n = {s. n ∈ req s}"
```

— The program

definition

```
request_act :: "(state*state) set"
where "request_act = {(s,s'). s' = s (/stop:=True, move:=False|)
& ~ stop s & floor s ∈ req s}"
```

definition

```
open_act :: "(state*state) set"
where "open_act =
{(s,s'). s' = s (/open :=True,
req := req s - {floor s},
move := True|)
& stop s & ~ open s & floor s ∈ req s
& ~(move s & s ∈ queueing)}"
```

definition

```
close_act :: "(state*state) set"
where "close_act = {(s,s'). s' = s (/open := False|) & open s}"
```

definition

```
req_up :: "(state*state) set"
where "req_up =
{(s,s'). s' = s (/stop :=False,
floor := floor s + 1,
up := True|)
& s ∈ (ready ∩ goingup)}"
```

definition

```
req_down :: "(state*state) set"
where "req_down =
{(s,s'). s' = s (/stop :=False,
floor := floor s - 1,
up := False|)
& s ∈ (ready ∩ goingdown)}"
```

definition

```

move_up :: "(state*state) set"
where "move_up =
      {(s,s'). s' = s (|floor := floor s + 1|)
       & ~ stop s & up s & floor s ∉ req s}"

```

definition

```

move_down :: "(state*state) set"
where "move_down =
      {(s,s'). s' = s (|floor := floor s - 1|)
       & ~ stop s & ~ up s & floor s ∉ req s}"

```

definition

```

button_press :: "(state*state) set"

```

— This action is omitted from prior treatments, which therefore are unrealistic: nobody asks the lift to do anything! But adding this action invalidates many of the existing progress arguments: various "ensures" properties fail. Maybe it should be constrained to only allow button presses in the current direction of travel, like in a real lift.

```

where "button_press =
      {(s,s'). ∃n. s' = s (|req := insert n (req s)|)
       & Min ≤ n & n ≤ Max}"

```

definition

```

Lift :: "state program"

```

— for the moment, we OMIT `button_press`

```

where "Lift = mk_total_program
      ({s. floor s = Min & ~ up s & move s & stop s &
       ~ open s & req s = {}},
      {request_act, open_act, close_act,
       req_up, req_down, move_up, move_down},
      UNIV)"

```

— Invariants

definition

```

bounded :: "state set"
where "bounded = {s. Min ≤ floor s & floor s ≤ Max}"

```

definition

```

open_stop :: "state set"
where "open_stop = {s. open s --> stop s}"

```

definition

```

open_move :: "state set"
where "open_move = {s. open s --> move s}"

```

definition

```

stop_floor :: "state set"
where "stop_floor = {s. stop s & ~ move s --> floor s ∈ req s}"

```

definition

```

moving_up :: "state set"
where "moving_up = {s. ~ stop s & up s -->
      (∃f. floor s ≤ f & f ≤ Max & f ∈ req s)}"

```

definition

```

moving_down :: "state set"
where "moving_down = {s. ~ stop s & ~ up s -->
      (∃f. Min ≤ f & f ≤ floor s & f ∈ req s)}"

```

definition

```

metric :: "[int,state] => int"
where "metric =
      (%n s. if floor s < n then (if up s then n - floor s
                                else (floor s - Min) + (n-Min))
      else
      if n < floor s then (if up s then (Max - floor s) + (Max-n)
                          else floor s - n)
      else 0)"

```

locale Floor =

```

fixes n
assumes Min_le_n [iff]: "Min ≤ n"
and n_le_Max [iff]: "n ≤ Max"

```

```

lemma not_mem_distinct: "[| x ∉ A; y ∈ A |] ==> x ≠ y"
<proof>

```

```

declare Lift_def [THEN def_prg_Init, simp]

```

```

declare request_act_def [THEN def_act_simp, simp]
declare open_act_def [THEN def_act_simp, simp]
declare close_act_def [THEN def_act_simp, simp]
declare req_up_def [THEN def_act_simp, simp]
declare req_down_def [THEN def_act_simp, simp]
declare move_up_def [THEN def_act_simp, simp]
declare move_down_def [THEN def_act_simp, simp]
declare button_press_def [THEN def_act_simp, simp]

```

```

declare above_def [THEN def_set_simp, simp]
declare below_def [THEN def_set_simp, simp]
declare queueing_def [THEN def_set_simp, simp]
declare goingup_def [THEN def_set_simp, simp]
declare goingdown_def [THEN def_set_simp, simp]
declare ready_def [THEN def_set_simp, simp]

```

```

declare bounded_def [simp]
      open_stop_def [simp]
      open_move_def [simp]
      stop_floor_def [simp]
      moving_up_def [simp]

```

`moving_down_def [simp]`

lemma `open_stop`: "Lift \in Always open_stop"
 \langle proof \rangle

lemma `stop_floor`: "Lift \in Always stop_floor"
 \langle proof \rangle

lemma `open_move`: "Lift \in Always open_move"
 \langle proof \rangle

lemma `moving_up`: "Lift \in Always moving_up"
 \langle proof \rangle

lemma `moving_down`: "Lift \in Always moving_down"
 \langle proof \rangle

lemma `bounded`: "Lift \in Always bounded"
 \langle proof \rangle

18.3 Progress

`declare moving_def [THEN def_set_simp, simp]`
`declare stopped_def [THEN def_set_simp, simp]`
`declare opened_def [THEN def_set_simp, simp]`
`declare closed_def [THEN def_set_simp, simp]`
`declare atFloor_def [THEN def_set_simp, simp]`
`declare Req_def [THEN def_set_simp, simp]`

The HUG'93 paper mistakenly omits the Req n from these!

lemma `E_thm01`: "Lift \in (stopped \cap atFloor n) LeadsTo (opened \cap atFloor n)"
 \langle proof \rangle

lemma `E_thm02`: "Lift \in (Req n \cap stopped - atFloor n) LeadsTo (Req n \cap opened - atFloor n)"
 \langle proof \rangle

lemma `E_thm03`: "Lift \in (Req n \cap opened - atFloor n) LeadsTo (Req n \cap closed - (atFloor n - queueing))"
 \langle proof \rangle

lemma `E_thm04`: "Lift \in (Req n \cap closed \cap (atFloor n - queueing)) LeadsTo (opened \cap atFloor n)"
 \langle proof \rangle

lemmas `linorder_leI = linorder_not_less [THEN iffD1]`

```

context Floor
begin

lemmas le_MinD = Min_le_n [THEN order_antisym]
  and Max_leD = n_le_Max [THEN [2] order_antisym]

declare le_MinD [dest!]
  and linorder_leI [THEN le_MinD, dest!]
  and Max_leD [dest!]
  and linorder_leI [THEN Max_leD, dest!]

lemma E_thm05c:
  "Lift ∈ (Req n ∩ closed - (atFloor n - queueing))
    LeadsTo ((closed ∩ goingup ∩ Req n) ∪
              (closed ∩ goingdown ∩ Req n))"
  <proof>

lemma lift_2: "Lift ∈ (Req n ∩ closed - (atFloor n - queueing))
  LeadsTo (moving ∩ Req n)"
  <proof>

declare if_split_asm [split]

lemma E_thm12a:
  "0 < N ==>
    Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩
             {s. floor s ∉ req s} ∩ {s. up s})
    LeadsTo
      (moving ∩ Req n ∩ {s. metric n s < N})"
  <proof>

lemma E_thm12b: "0 < N ==>
    Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩
             {s. floor s ∉ req s} - {s. up s})
    LeadsTo (moving ∩ Req n ∩ {s. metric n s < N})"
  <proof>

lemma lift_4:
  "0 < N ==> Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩
                     {s. floor s ∉ req s}) LeadsTo
    (moving ∩ Req n ∩ {s. metric n s < N})"
  <proof>

```


lemma *E_thm16a*: " $0 < N$
 $\implies \text{Lift} \in (\text{closed} \cap \text{Req } n \cap \{s. \text{metric } n \ s = N\} \cap \text{goingup}) \text{ LeadsTo}$
 $(\text{moving} \cap \text{Req } n \cap \{s. \text{metric } n \ s < N\})$ "
 <proof>

lemma *E_thm16b*: " $0 < N \implies$
 $\text{Lift} \in (\text{closed} \cap \text{Req } n \cap \{s. \text{metric } n \ s = N\} \cap \text{goingdown}) \text{ LeadsTo}$
 $(\text{moving} \cap \text{Req } n \cap \{s. \text{metric } n \ s < N\})$ "
 <proof>

lemma *E_thm16c*:
 $0 < N \implies \text{Req } n \cap \{s. \text{metric } n \ s = N\} \subseteq \text{goingup} \cup \text{goingdown}$ "
 <proof>

lemma *lift_5*:
 $0 < N \implies \text{Lift} \in (\text{closed} \cap \text{Req } n \cap \{s. \text{metric } n \ s = N\}) \text{ LeadsTo}$
 $(\text{moving} \cap \text{Req } n \cap \{s. \text{metric } n \ s < N\})$ "
 <proof>

lemma *metric_eq_OD [dest]*:
 $[| \text{metric } n \ s = 0; \text{Min} \leq \text{floor } s; \text{floor } s \leq \text{Max} |] \implies \text{floor } s =$
 n "
 <proof>

lemma *E_thm11*: " $\text{Lift} \in (\text{moving} \cap \text{Req } n \cap \{s. \text{metric } n \ s = 0\}) \text{ LeadsTo}$
 $(\text{stopped} \cap \text{atFloor } n)$ "
 <proof>

lemma *E_thm13*:
 $\text{Lift} \in (\text{moving} \cap \text{Req } n \cap \{s. \text{metric } n \ s = N\} \cap \{s. \text{floor } s \in \text{req } s\})$
 $\text{ LeadsTo } (\text{stopped} \cap \text{Req } n \cap \{s. \text{metric } n \ s = N\} \cap \{s. \text{floor } s \in \text{req } s\})$ "
 <proof>

lemma *E_thm14*: " $0 < N \implies$

```

    Lift ∈
      (stopped ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})
      LeadsTo (opened ∩ Req n ∩ {s. metric n s = N})"
  ⟨proof⟩

```

```

lemma E_thm15: "Lift ∈ (opened ∩ Req n ∩ {s. metric n s = N})
  LeadsTo (closed ∩ Req n ∩ {s. metric n s = N})"
  ⟨proof⟩

```

```

lemma lift_3_Req: "0 < N ==>
  Lift ∈
    (moving ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})
    LeadsTo (moving ∩ Req n ∩ {s. metric n s < N})"
  ⟨proof⟩

```

```

lemma Always_nonneg: "Lift ∈ Always {s. 0 ≤ metric n s}"
  ⟨proof⟩

```

```

lemmas R_thm11 = Always_LeadsTo_weaken [OF Always_nonneg E_thm11]

```

```

lemma lift_3: "Lift ∈ (moving ∩ Req n) LeadsTo (stopped ∩ atFloor n)"
  ⟨proof⟩

```

```

lemma lift_1: "Lift ∈ (Req n) LeadsTo (opened ∩ atFloor n)"
  ⟨proof⟩

```

end

end

```

theory Mutex imports "../UNITY_Main" begin

```

```

record state =
  p :: bool
  m :: int
  n :: int
  u :: bool
  v :: bool

```

```

type_synonym command = "(state*state) set"

```

```

definition U0 :: command
  where "U0 = {(s,s'). s' = s (/u:=True, m:=1/) & m s = 0}"

```

```

definition U1 :: command
  where "U1 = {(s,s'). s' = s (|p:=v s, m:=2|) & m s = 1}"

definition U2 :: command
  where "U2 = {(s,s'). s' = s (|m:=3|) & ~ p s & m s = 2}"

definition U3 :: command
  where "U3 = {(s,s'). s' = s (|u:=False, m:=4|) & m s = 3}"

definition U4 :: command
  where "U4 = {(s,s'). s' = s (|p:=True, m:=0|) & m s = 4}"

definition V0 :: command
  where "V0 = {(s,s'). s' = s (|v:=True, n:=1|) & n s = 0}"

definition V1 :: command
  where "V1 = {(s,s'). s' = s (|p:=~ u s, n:=2|) & n s = 1}"

definition V2 :: command
  where "V2 = {(s,s'). s' = s (|n:=3|) & p s & n s = 2}"

definition V3 :: command
  where "V3 = {(s,s'). s' = s (|v:=False, n:=4|) & n s = 3}"

definition V4 :: command
  where "V4 = {(s,s'). s' = s (|p:=False, n:=0|) & n s = 4}"

definition Mutex :: "state program"
  where "Mutex = mk_total_program
    ({s. ~ u s & ~ v s & m s = 0 & n s = 0},
     {U0, U1, U2, U3, U4, V0, V1, V2, V3, V4},
     UNIV)"

definition IU :: "state set"
  where "IU = {s. (u s = (1 ≤ m s & m s ≤ 3)) & (m s = 3 --> ~ p s)}"

definition IV :: "state set"
  where "IV = {s. (v s = (1 ≤ n s & n s ≤ 3)) & (n s = 3 --> p s)}"

definition bad_IU :: "state set"
  where "bad_IU = {s. (u s = (1 ≤ m s & m s ≤ 3)) &
    (3 ≤ m s & m s ≤ 4 --> ~ p s)}"

declare Mutex_def [THEN def_prg_Init, simp]

```

```

declare U0_def [THEN def_act_simp, simp]
declare U1_def [THEN def_act_simp, simp]
declare U2_def [THEN def_act_simp, simp]
declare U3_def [THEN def_act_simp, simp]
declare U4_def [THEN def_act_simp, simp]
declare V0_def [THEN def_act_simp, simp]
declare V1_def [THEN def_act_simp, simp]
declare V2_def [THEN def_act_simp, simp]
declare V3_def [THEN def_act_simp, simp]
declare V4_def [THEN def_act_simp, simp]

declare IU_def [THEN def_set_simp, simp]
declare IV_def [THEN def_set_simp, simp]
declare bad_IU_def [THEN def_set_simp, simp]

lemma IU: "Mutex  $\in$  Always IU"
<proof>

lemma IV: "Mutex  $\in$  Always IV"
<proof>

lemma mutual_exclusion: "Mutex  $\in$  Always {s.  $\sim$  (m s = 3 & n s = 3)}"
<proof>

lemma "Mutex  $\in$  Always bad_IU"
<proof>

lemma eq_123: "((1::int)  $\leq$  i & i  $\leq$  3) = (i = 1 | i = 2 | i = 3)"
<proof>

lemma U_F0: "Mutex  $\in$  {s. m s=2} Unless {s. m s=3}"
<proof>

lemma U_F1: "Mutex  $\in$  {s. m s=1} LeadsTo {s. p s = v s & m s = 2}"
<proof>

lemma U_F2: "Mutex  $\in$  {s.  $\sim$  p s & m s = 2} LeadsTo {s. m s = 3}"
<proof>

lemma U_F3: "Mutex  $\in$  {s. m s = 3} LeadsTo {s. p s}"
<proof>

lemma U_lemma2: "Mutex  $\in$  {s. m s = 2} LeadsTo {s. p s}"
<proof>

lemma U_lemma1: "Mutex  $\in$  {s. m s = 1} LeadsTo {s. p s}"

```

<proof>

lemma *U_lemma123*: "Mutex \in {s. $1 \leq m\ s \ \& \ m\ s \leq 3$ } LeadsTo {s. p s}"
<proof>

lemma *u_Leadsto_p*: "Mutex \in {s. u s} LeadsTo {s. p s}"
<proof>

lemma *V_F0*: "Mutex \in {s. n s=2} Unless {s. n s=3}"
<proof>

lemma *V_F1*: "Mutex \in {s. n s=1} LeadsTo {s. p s = (\sim u s) $\&$ n s = 2}"
<proof>

lemma *V_F2*: "Mutex \in {s. p s $\&$ n s = 2} LeadsTo {s. n s = 3}"
<proof>

lemma *V_F3*: "Mutex \in {s. n s = 3} LeadsTo {s. \sim p s}"
<proof>

lemma *V_lemma2*: "Mutex \in {s. n s = 2} LeadsTo {s. \sim p s}"
<proof>

lemma *V_lemma1*: "Mutex \in {s. n s = 1} LeadsTo {s. \sim p s}"
<proof>

lemma *V_lemma123*: "Mutex \in {s. $1 \leq n\ s \ \& \ n\ s \leq 3$ } LeadsTo {s. \sim p s}"
<proof>

lemma *v_Leadsto_not_p*: "Mutex \in {s. v s} LeadsTo {s. \sim p s}"
<proof>

lemma *m1_Leadsto_3*: "Mutex \in {s. m s = 1} LeadsTo {s. m s = 3}"
<proof>

lemma *n1_Leadsto_3*: "Mutex \in {s. n s = 1} LeadsTo {s. n s = 3}"
<proof>

end

theory *Reach* imports "../UNITY_Main" begin

```

typedecl vertex

type_synonym state = "vertex=>bool"

consts
  init :: "vertex"

  edges :: "(vertex*vertex) set"

definition asgt :: "[vertex,vertex] => (state*state) set"
  where "asgt u v = {(s,s'). s' = s(v:= s u | s v)}"

definition Rprg :: "state program"
  where "Rprg = mk_total_program ({%v. v=init},  $\bigcup_{(u,v)\in\text{edges.}} \{\text{asgt } u \ v\}$ ,
  UNIV)"

definition reach_invariant :: "state set"
  where "reach_invariant = {s. ( $\forall v. s \ v \ \rightarrow (init, v) \in \text{edges}^*$ ) & s init}"

definition fixedpoint :: "state set"
  where "fixedpoint = {s.  $\forall (u,v)\in\text{edges. } s \ u \ \rightarrow s \ v$ }"

definition metric :: "state => nat"
  where "metric s = card {v.  $\sim s \ v$ }"

*We assume that the set of vertices is finite

axiomatization where
  finite_graph: "finite (UNIV :: vertex set)"

lemma ifE [elim!]:
  "[| if P then Q else R;
   [| P; Q |] ==> S;
   [| ~ P; R |] ==> S |] ==> S"
  <proof>

declare Rprg_def [THEN def_prg_Init, simp]

declare asgt_def [THEN def_act_simp, simp]

All vertex sets are finite
declare finite_subset [OF subset_UNIV finite_graph, iff]

declare reach_invariant_def [THEN def_set_simp, simp]

lemma reach_invariant: "Rprg  $\in$  Always reach_invariant"
  <proof>

```

```

lemma fixedpoint_invariant_correct:
  "fixedpoint  $\cap$  reach_invariant = { %v. (init, v)  $\in$  edges* }"
  <proof>

lemma lemma1:
  "FP Rprg  $\subseteq$  fixedpoint"
  <proof>

lemma lemma2:
  "fixedpoint  $\subseteq$  FP Rprg"
  <proof>

lemma FP_fixedpoint: "FP Rprg = fixedpoint"
  <proof>

lemma Compl_fixedpoint: "- fixedpoint = ( $\bigcup_{(u,v) \in \text{edges}} \{s. s \ u \ \& \ \sim \ s \ v\}$ )"
  <proof>

lemma Diff_fixedpoint:
  "A - fixedpoint = ( $\bigcup_{(u,v) \in \text{edges}} A \cap \{s. s \ u \ \& \ \sim \ s \ v\}$ )"
  <proof>

lemma Suc_metric: "~ s x ==> Suc (metric (s(x:=True))) = metric s"
  <proof>

lemma metric_less [intro!]: "~ s x ==> metric (s(x:=True)) < metric s"
  <proof>

lemma metric_le: "metric (s(y:=s x | s y))  $\leq$  metric s"
  <proof>

lemma LeadsTo_Diff_fixedpoint:
  "Rprg  $\in$  ((metric-' $\{m\}$ ) - fixedpoint) LeadsTo (metric-' $\{\text{lessThan } m\}$ )"
  <proof>

lemma LeadsTo_Un_fixedpoint:
  "Rprg  $\in$  (metric-' $\{m\}$ ) LeadsTo (metric-' $\{\text{lessThan } m\} \cup \text{fixedpoint}$ )"
  <proof>

lemma LeadsTo_fixedpoint: "Rprg  $\in$  UNIV LeadsTo fixedpoint"
  <proof>

lemma LeadsTo_correct: "Rprg  $\in$  UNIV LeadsTo { %v. (init, v)  $\in$  edges* }"

```

<proof>

end

theory Reachability imports "../Detects" Reach begin

type_synonym edge = "vertex * vertex"

record state =

 reach :: "vertex => bool"
 nmsg :: "edge => nat"

consts root :: "vertex"

 E :: "edge set"

 V :: "vertex set"

inductive_set REACHABLE :: "edge set"

 where

 base: " $v \in V \implies ((v,v) \in REACHABLE)$ "

 | step: " $((u,v) \in REACHABLE) \ \& \ (v,w) \in E \implies ((u,w) \in REACHABLE)$ "

definition reachable :: "vertex => state set" where

 "reachable p == {s. reach s p}"

definition nmsg_eq :: "nat => edge => state set" where

 "nmsg_eq k == %e. {s. nmsg s e = k}"

definition nmsg_gt :: "nat => edge => state set" where

 "nmsg_gt k == %e. {s. k < nmsg s e}"

definition nmsg_gte :: "nat => edge => state set" where

 "nmsg_gte k == %e. {s. k \leq nmsg s e}"

definition nmsg_lte :: "nat => edge => state set" where

 "nmsg_lte k == %e. {s. nmsg s e \leq k}"

definition final :: "state set" where

 "final == $(\bigcap_{v \in V. \text{reachable } v} \iff \{s. (root, v) \in REACHABLE\}) \cap$
 $(\bigcap ((nmsg_eq\ 0) \ ' E))"$

axiomatization

where

 Graph1: "root $\in V$ " and

 Graph2: " $(v,w) \in E \implies (v \in V) \ \& \ (w \in V)$ " and

 MA1: "F \in Always (reachable root)" and

 MA2: " $v \in V \implies F \in$ Always $(-\text{reachable } v \cup \{s. ((root,v) \in REACHABLE)\})"$

and

 MA3: " $[/v \in V; w \in V/] \implies F \in$ Always $(-(nmsg_gt\ 0\ (v,w)) \cup (\text{reachable } v))"$ and


```

MA4: "(v,w) ∈ E ==>
      F ∈ Always (-(reachable v) ∪ (nmsg_gt 0 (v,w)) ∪ (reachable w))"
and
MA5: "[|v ∈ V; w ∈ V|]
      ==> F ∈ Always (nmsg_gte 0 (v,w) ∩ nmsg_lte (Suc 0) (v,w))" and
MA6: "[|v ∈ V|] ==> F ∈ Stable (reachable v)" and
MA6b: "[|v ∈ V; w ∈ W|] ==> F ∈ Stable (reachable v ∩ nmsg_lte k (v,w))"
and
MA7: "[|v ∈ V; w ∈ V|] ==> F ∈ UNIV LeadsTo nmsg_eq 0 (v,w)"

lemmas E_imp_in_V_L = Graph2 [THEN conjunct1]
lemmas E_imp_in_V_R = Graph2 [THEN conjunct2]

lemma lemma2:
  "(v,w) ∈ E ==> F ∈ reachable v LeadsTo nmsg_eq 0 (v,w) ∩ reachable v"
⟨proof⟩

lemma Induction_base: "(v,w) ∈ E ==> F ∈ reachable v LeadsTo reachable w"
⟨proof⟩

lemma REACHABLE_LeadsTo_reachable:
  "(v,w) ∈ REACHABLE ==> F ∈ reachable v LeadsTo reachable w"
⟨proof⟩

lemma Detects_part1: "F ∈ {s. (root,v) ∈ REACHABLE} LeadsTo reachable v"
⟨proof⟩

lemma Reachability_Detected:
  "v ∈ V ==> F ∈ (reachable v) Detects {s. (root,v) ∈ REACHABLE}"
⟨proof⟩

lemma LeadsTo_Reachability:
  "v ∈ V ==> F ∈ UNIV LeadsTo (reachable v <==> {s. (root,v) ∈ REACHABLE})"
⟨proof⟩

lemma Eq_lemma1:
  "(reachable v <==> {s. (root,v) ∈ REACHABLE}) =
   {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))}"
⟨proof⟩

```

lemma Eq_lemma2:
 "(reachable v <==> (if (root,v) ∈ REACHABLE then UNIV else {})) =
 {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))}"
 <proof>

lemma final_lemma1:
 "($\bigcap v \in V. \bigcap w \in V. \{s. ((s \in \text{reachable } v) = ((\text{root},v) \in \text{REACHABLE}))$)
 &
 $s \in \text{nmsg_eq } 0 (v,w)\}$)
 $\subseteq \text{final}$ "
 <proof>

lemma final_lemma2:
 "E ≠ {}
 ==> ($\bigcap v \in V. \bigcap e \in E. \{s. ((s \in \text{reachable } v) = ((\text{root},v) \in \text{REACHABLE}))\}$)
 $\cap \text{nmsg_eq } 0 e) \subseteq \text{final}$ "
 <proof>

lemma final_lemma3:
 "E ≠ {}
 ==> ($\bigcap v \in V. \bigcap e \in E.$
 (reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 e)
 $\subseteq \text{final}$ "
 <proof>

lemma final_lemma4:
 "E ≠ {}
 ==> ($\bigcap v \in V. \bigcap e \in E.$
 {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))} ∩ nmsg_eq 0
 e)
 = final"
 <proof>

lemma final_lemma5:
 "E ≠ {}
 ==> ($\bigcap v \in V. \bigcap e \in E.$
 (reachable v) <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 e)
 = final"
 <proof>

lemma final_lemma6:
 "($\bigcap v \in V. \bigcap w \in V.$
 (reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 (v,w))
 $\subseteq \text{final}$ "
 <proof>

lemma *final_lemma7*:

```
"final =
  (∩ v ∈ V. ∩ w ∈ V.
    ((reachable v) <==> {s. (root,v) ∈ REACHABLE}) ∩
    (-{s. (v,w) ∈ E} ∪ (nmsg_eq 0 (v,w))))"
⟨proof⟩
```

lemma *not_REACHABLE_imp_Stable_not_reachable*:

```
"[| v ∈ V; (root,v) ∉ REACHABLE |] ==> F ∈ Stable (- reachable v)"
⟨proof⟩
```

lemma *Stable_reachable_EQ_R*:

```
"v ∈ V ==> F ∈ Stable (reachable v <==> {s. (root,v) ∈ REACHABLE})"
⟨proof⟩
```

lemma *lemma4*:

```
"((nmsg_gte 0 (v,w) ∩ nmsg_lte (Suc 0) (v,w)) ∩
  (- nmsg_gt 0 (v,w) ∪ A))
 ⊆ A ∪ nmsg_eq 0 (v,w)"
⟨proof⟩
```

lemma *lemma5*:

```
"reachable v ∩ nmsg_eq 0 (v,w) =
  ((nmsg_gte 0 (v,w) ∩ nmsg_lte (Suc 0) (v,w)) ∩
   (reachable v ∩ nmsg_lte 0 (v,w)))"
⟨proof⟩
```

lemma *lemma6*:

```
"- nmsg_gt 0 (v,w) ∪ reachable v ⊆ nmsg_eq 0 (v,w) ∪ reachable v"
⟨proof⟩
```

lemma *Always_reachable_OR_nmsg_0*:

```
"[| v ∈ V; w ∈ V |] ==> F ∈ Always (reachable v ∪ nmsg_eq 0 (v,w))"
⟨proof⟩
```

lemma *Stable_reachable_AND_nmsg_0*:

```
"[| v ∈ V; w ∈ V |] ==> F ∈ Stable (reachable v ∩ nmsg_eq 0 (v,w))"
⟨proof⟩
```

lemma *Stable_nmsg_0_OR_reachable*:

```
"[| v ∈ V; w ∈ V |] ==> F ∈ Stable (nmsg_eq 0 (v,w) ∪ reachable v)"
⟨proof⟩
```

lemma *not_REACHABLE_imp_Stable_not_reachable_AND_nmsg_0*:

```
"[| v ∈ V; w ∈ V; (root,v) ∉ REACHABLE |]
```

```

    ==> F ∈ Stable (- reachable v ∩ nmsg_eq 0 (v,w))"
  <proof>

lemma Stable_reachable_EQ_R_AND_nmsg_0:
  "[| v ∈ V; w ∈ V |]
  ==> F ∈ Stable ((reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩
    nmsg_eq 0 (v,w))"
  <proof>

lemma UNIV_lemma: "UNIV ⊆ (⋂ v ∈ V. UNIV)"
  <proof>

lemmas UNIV_LeadsTo_completion =
  LeadsTo_weaken_L [OF Finite_stable_completion UNIV_lemma]

lemma LeadsTo_final_E_empty: "E={} ==> F ∈ UNIV LeadsTo final"
  <proof>

lemma Leadsto_reachability_AND_nmsg_0:
  "[| v ∈ V; w ∈ V |]
  ==> F ∈ UNIV LeadsTo
    ((reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 (v,w))"
  <proof>

lemma LeadsTo_final_E_NOT_empty: "E≠{} ==> F ∈ UNIV LeadsTo final"
  <proof>

lemma LeadsTo_final: "F ∈ UNIV LeadsTo final"
  <proof>

lemma Stable_final_E_empty: "E={} ==> F ∈ Stable final"
  <proof>

lemma Stable_final_E_NOT_empty: "E≠{} ==> F ∈ Stable final"
  <proof>

lemma Stable_final: "F ∈ Stable final"
  <proof>

end

```

19 Analyzing the Needham-Schroeder Public-Key Protocol in UNITY

```
theory NSP_Bad imports "HOL-Auth.Public" "../UNITY_Main" begin
```

This is the flawed version, vulnerable to Lowe's attack. From page 260 of Burrows, Abadi and Needham. A Logic of Authentication. Proc. Royal Soc. 426 (1989).

```
type_synonym state = "event list"
```

definition

```
Fake :: "(state*state) set"
where "Fake = {(s,s').
         $\exists B X. s' = \text{Says Spy } B \ X \ \# \ s$ 
        &  $X \in \text{synth } (\text{analz } (\text{spies } s))\}"$ 
```

definition

```
NS1 :: "(state*state) set"
where "NS1 = {(s1,s').
         $\exists A1 \ B \ NA.$ 
         $s' = \text{Says } A1 \ B \ (\text{Crypt } (\text{pubK } B) \ \{\text{Nonce } NA, \text{Agent } A1\}) \ \# \ s1$ 
        &  $\text{Nonce } NA \notin \text{used } s1\}"$ 
```

definition

```
NS2 :: "(state*state) set"
where "NS2 = {(s2,s').
         $\exists A' \ A2 \ B \ NA \ NB.$ 
         $s' = \text{Says } B \ A2 \ (\text{Crypt } (\text{pubK } A2) \ \{\text{Nonce } NA, \text{Nonce } NB\}) \ \# \ s2$ 
        &  $\text{Says } A' \ B \ (\text{Crypt } (\text{pubK } B) \ \{\text{Nonce } NA, \text{Agent } A2\}) \in \text{set } s2$ 
        &  $\text{Nonce } NB \notin \text{used } s2\}"$ 
```

definition

```
NS3 :: "(state*state) set"
where "NS3 = {(s3,s').
         $\exists A3 \ B' \ B \ NA \ NB.$ 
         $s' = \text{Says } A3 \ B \ (\text{Crypt } (\text{pubK } B) \ (\text{Nonce } NB)) \ \# \ s3$ 
        &  $\text{Says } A3 \ B \ (\text{Crypt } (\text{pubK } B) \ \{\text{Nonce } NA, \text{Agent } A3\}) \in \text{set } s3$ 
        &  $\text{Says } B' \ A3 \ (\text{Crypt } (\text{pubK } A3) \ \{\text{Nonce } NA, \text{Nonce } NB\}) \in \text{set } s3\}"$ 
```

definition Nprg :: "state program" where

```
"Nprg = mk_total_program({[]}, {Fake, NS1, NS2, NS3}, UNIV)"
```

```
declare spies_partsEs [elim]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]
```

For other theories, e.g. Mutex and Lift, using [iff] slows proofs down. Here, it facilitates re-use of the Auth proofs.

```

declare Fake_def [THEN def_act_simp, iff]
declare NS1_def [THEN def_act_simp, iff]
declare NS2_def [THEN def_act_simp, iff]
declare NS3_def [THEN def_act_simp, iff]

declare Nprg_def [THEN def_prg_Init, simp]

```

A "possibility property": there are traces that reach the end. Replace by LEAD-STO proof!

```

lemma "A ≠ B ==>
  ∃ NB. ∃ s ∈ reachable Nprg. Says A B (Crypt (pubK B) (Nonce NB)) ∈ set
  s"
<proof>

```

19.1 Inductive Proofs about *ns_public*

```

lemma ns_constrainsI:
  "(!act s s'. [| act ∈ {Id, Fake, NS1, NS2, NS3};
                (s,s') ∈ act; s ∈ A |] ==> s' ∈ A)"
  ==> Nprg ∈ A co A'"
<proof>

```

This ML code does the inductions directly.

<ML>

Converts invariants into statements about reachable states

```

lemmas Always_Collect_reachableD =
  Always_includes_reachable [THEN subsetD, THEN CollectD]

```

Spy never sees another agent's private key! (unless it's bad at start)

```

lemma Spy_see_priK:
  "Nprg ∈ Always {s. (Key (priK A) ∈ parts (spies s)) = (A ∈ bad)}"
<proof>
declare Spy_see_priK [THEN Always_Collect_reachableD, simp]

```

```

lemma Spy_analz_priK:
  "Nprg ∈ Always {s. (Key (priK A) ∈ analz (spies s)) = (A ∈ bad)}"
<proof>
declare Spy_analz_priK [THEN Always_Collect_reachableD, simp]

```

19.2 Authenticity properties obtained from NS2

It is impossible to re-use a nonce in both NS1 and NS2 provided the nonce is secret. (Honest users generate fresh nonces.)

```

lemma no_nonce_NS1_NS2:
  "Nprg
  ∈ Always {s. Crypt (pubK C) {NA', Nonce NA} ∈ parts (spies s) -->
              Crypt (pubK B) {Nonce NA, Agent A} ∈ parts (spies s) -->
              Nonce NA ∈ analz (spies s)}"
<proof>

```

Adding it to the claset slows down proofs...

```
lemmas no_nonce_NS1_NS2_reachable =
  no_nonce_NS1_NS2 [THEN Always_Collect_reachableD, rule_format]
```

Unicity for NS1: nonce NA identifies agents A and B

```
lemma unique_NA_lemma:
  "Nprg
  ∈ Always {s. Nonce NA ∉ analz (spies s) -->
    Crypt(pubK B) {Nonce NA, Agent A} ∈ parts(spies s) -->
    Crypt(pubK B') {Nonce NA, Agent A'} ∈ parts(spies s) -->
    A=A' & B=B'}"
```

<proof>

Unicity for NS1: nonce NA identifies agents A and B

```
lemma unique_NA:
  "[| Crypt(pubK B) {Nonce NA, Agent A} ∈ parts(spies s);
    Crypt(pubK B') {Nonce NA, Agent A'} ∈ parts(spies s);
    Nonce NA ∉ analz (spies s);
    s ∈ reachable Nprg |]
  ==> A=A' & B=B'"
```

<proof>

Secrecy: Spy does not see the nonce sent in msg NS1 if A and B are secure

```
lemma Spy_not_see_NA:
  "[| A ∉ bad; B ∉ bad |]
  ==> Nprg ∈ Always
    {s. Says A B (Crypt(pubK B) {Nonce NA, Agent A}) ∈ set s
      --> Nonce NA ∉ analz (spies s)}"
```

<proof>

Authentication for A: if she receives message 2 and has used NA to start a run, then B has sent message 2.

```
lemma A_trusts_NS2:
  "[| A ∉ bad; B ∉ bad |]
  ==> Nprg ∈ Always
    {s. Says A B (Crypt(pubK B) {Nonce NA, Agent A}) ∈ set s &
      Crypt(pubK A) {Nonce NA, Nonce NB} ∈ parts (knows Spy s)
      --> Says B A (Crypt(pubK A) {Nonce NA, Nonce NB}) ∈ set s}"
```

<proof>

If the encrypted message appears then it originated with Alice in NS1

```
lemma B_trusts_NS1:
  "Nprg ∈ Always
    {s. Nonce NA ∉ analz (spies s) -->
      Crypt (pubK B) {Nonce NA, Agent A} ∈ parts (spies s)
      --> Says A B (Crypt (pubK B) {Nonce NA, Agent A}) ∈ set s}"
```

<proof>

19.3 Authenticity properties obtained from NS2

Unicity for NS2: nonce NB identifies nonce NA and agent A. Proof closely follows that of *unique_NA*.

```

lemma unique_NB_lemma:
  "Nprg
   ∈ Always {s. Nonce NB ∉ analz (spies s) -->
              Crypt (pubK A) {Nonce NA, Nonce NB} ∈ parts (spies s) -->
              Crypt (pubK A') {Nonce NA', Nonce NB} ∈ parts (spies s) -->
              A=A' & NA=NA'}"
  <proof>

```

```

lemma unique_NB:
  "[| Crypt (pubK A) {Nonce NA, Nonce NB} ∈ parts (spies s);
    Crypt (pubK A') {Nonce NA', Nonce NB} ∈ parts (spies s);
    Nonce NB ∉ analz (spies s);
    s ∈ reachable Nprg |]
   ==> A=A' & NA=NA'"
  <proof>

```

NB remains secret PROVIDED Alice never responds with round 3

```

lemma Spy_not_see_NB:
  "[| A ∉ bad; B ∉ bad |]
   ==> Nprg ∈ Always
      {s. Says B A (Crypt (pubK A) {Nonce NA, Nonce NB}) ∈ set s &
        (∀ C. Says A C (Crypt (pubK C) (Nonce NB)) ∉ set s)
        --> Nonce NB ∉ analz (spies s)}"
  <proof>

```

Authentication for B: if he receives message 3 and has used NB in message 2, then A has sent message 3—to somebody....

```

lemma B_trusts_NS3:
  "[| A ∉ bad; B ∉ bad |]
   ==> Nprg ∈ Always
      {s. Crypt (pubK B) (Nonce NB) ∈ parts (spies s) &
        Says B A (Crypt (pubK A) {Nonce NA, Nonce NB}) ∈ set s
        --> (∃ C. Says A C (Crypt (pubK C) (Nonce NB)) ∈ set s)}"
  <proof>

```

Can we strengthen the secrecy theorem? NO

```

lemma "[| A ∉ bad; B ∉ bad |]
   ==> Nprg ∈ Always
      {s. Says B A (Crypt (pubK A) {Nonce NA, Nonce NB}) ∈ set s
        --> Nonce NB ∉ analz (spies s)}"
  <proof>

```

end

theory Handshake imports "../UNITY_Main" **begin**

```

record state =
  BB :: bool

```



```

NF :: nat
NG :: nat

definition
  cmdF :: "(state*state) set"
  where "cmdF = {(s,s'). s' = s (|NF:= Suc(NF s), BB:=False|) & BB s}"

definition
  F :: "state program"
  where "F = mk_total_program ({s. NF s = 0 & BB s}, {cmdF}, UNIV)"

definition
  cmdG :: "(state*state) set"
  where "cmdG = {(s,s'). s' = s (|NG:= Suc(NG s), BB:=True|) & ~ BB s}"

definition
  G :: "state program"
  where "G = mk_total_program ({s. NG s = 0 & BB s}, {cmdG}, UNIV)"

definition
  invFG :: "state set"
  where "invFG = {s. NG s <= NF s & NF s <= Suc (NG s) & (BB s = (NF s = NG s))}"

declare F_def [THEN def_prg_Init, simp]
        G_def [THEN def_prg_Init, simp]

        cmdF_def [THEN def_act_simp, simp]
        cmdG_def [THEN def_act_simp, simp]

        invFG_def [THEN def_set_simp, simp]

lemma invFG: "(F  $\sqcup$  G)  $\in$  Always invFG"
<proof>

lemma lemma2_1: "(F  $\sqcup$  G)  $\in$  ({s. NF s = k} - {s. BB s}) LeadsTo
  ({s. NF s = k} Int {s. BB s})"
<proof>

lemma lemma2_2: "(F  $\sqcup$  G)  $\in$  ({s. NF s = k} Int {s. BB s}) LeadsTo
  {s. k < NF s}"
<proof>

lemma progress: "(F  $\sqcup$  G)  $\in$  UNIV LeadsTo {s. m < NF s}"
<proof>

end

```

20 A Family of Similar Counters: Original Version

```

theory Counter imports "../UNITY_Main" begin

datatype name = C | c nat
type_synonym state = "name=>int"

primrec sum :: "[nat,state]=>int" where

  "sum 0 s = 0"
| "sum (Suc i) s = s (c i) + sum i s"

primrec sumj :: "[nat, nat, state]=>int" where
  "sumj 0 i s = 0"
| "sumj (Suc n) i s = (if n=i then sum n s else s (c n) + sumj n i s)"

type_synonym command = "(state*state)set"

definition a :: "nat=>command" where
  "a i = {(s, s'). s'=s(c i:= s (c i) + 1, C:= s C + 1)}"

definition Component :: "nat => state program" where
  "Component i =
    mk_total_program({s. s C = 0 & s (c i) = 0}, {a i},
       $\bigcup G \in \text{preserves } (\%s. s (c i)). \text{ Acts } G$ )"

declare Component_def [THEN def_prg_Init, simp]
declare a_def [THEN def_act_simp, simp]

lemma sum_upd_gt: "I<n ==> sum I (s(c n := x)) = sum I s"
  <proof>

lemma sum_upd_eq: "sum I (s(c I := x)) = sum I s"
  <proof>

lemma sum_upd_C: "sum I (s(C := x)) = sum I s"
  <proof>

lemma sumj_upd_ci: "sumj I i (s(c i := x)) = sumj I i s"
  <proof>

lemma sumj_upd_C: "sumj I i (s(C := x)) = sumj I i s"
  <proof>

lemma sumj_sum_gt: "I<i ==> sumj I i s = sum I s"
  <proof>

lemma sumj_sum_eq: "(sumj I I s = sum I s)"

```

```

    <proof>

lemma sum_sumj: "i < I ==> sum I s = s (c i) + sumj I i s"
  <proof>

lemma p2: "Component i ∈ stable {s. s C = s (c i) + k}"
  <proof>

lemma p3: "Component i ∈ stable {s. ∀ v. v ≠ c i & v ≠ C --> s v = k v}"
  <proof>

lemma p2_p3_lemma1:
  "(∀ k. Component i ∈ stable ({s. s C = s (c i) + sumj I i k}
    ∩ {s. ∀ v. v ≠ c i & v ≠ C --> s v = k v}))
  = (Component i ∈ stable {s. s C = s (c i) + sumj I i s})"
  <proof>

lemma p2_p3_lemma2:
  "∀ k. Component i ∈ stable ({s. s C = s (c i) + sumj I i k} Int
    {s. ∀ v. v ≠ c i & v ≠ C --> s v = k v})"
  <proof>

lemma p2_p3: "Component i ∈ stable {s. s C = s (c i) + sumj I i s}"
  <proof>

lemma sum_0': "(∧ i. i < I ==> s (c i) = 0) ==> sum I s = 0"
  <proof>

lemma safety:
  "0 < I ==> (∏ i ∈ {i. i < I}. Component i) ∈ invariant {s. s C = sum I s}"
  <proof>

end

```

21 A Family of Similar Counters: Version with Compatibility

```

theory CounterC imports "../UNITY_Main" begin

typedecl state

consts
  C :: "state=>int"
  c :: "state=>nat=>int"

primrec sum :: "[nat,state]=>int" where

```

```

"sum 0 s = 0"
| "sum (Suc i) s = (c s) i + sum i s"

primrec sumj :: "[nat, nat, state]=>int" where
  "sumj 0 i s = 0"
| "sumj (Suc n) i s = (if n=i then sum n s else (c s) n + sumj n i s)"

type_synonym command = "(state*state)set"

definition a :: "nat=>command" where
  "a i = {(s, s'). (c s') i = (c s) i + 1 & (C s') = (C s) + 1}"

definition Component :: "nat => state program" where
  "Component i = mk_total_program({s. C s = 0 & (c s) i = 0},
    {a i},
     $\bigcup G \in \text{preserves } (\%s. (c s) i). \text{ Acts } G$ )"

declare Component_def [THEN def_prg_Init, simp]
declare Component_def [THEN def_prg_AllowedActs, simp]
declare a_def [THEN def_act_simp, simp]

lemma sum_sumj_eq1: "I<i ==> sum I s = sumj I i s"
  <proof>

lemma sum_sumj_eq2: "i<I ==> sum I s = c s i + sumj I i s"
  <proof>

lemma sum_ext: " $(\bigwedge i. i<I \implies c s' i = c s i) \implies \text{sum } I s' = \text{sum } I s$ "
  <proof>

lemma sumj_ext: " $(\bigwedge j. j<I \implies j \neq i \implies c s' j = c s j) \implies \text{sumj } I i s' = \text{sumj } I i s$ "
  <proof>

lemma sum0: " $(\bigwedge i. i<I \implies c s i = 0) \implies \text{sum } I s = 0$ "
  <proof>

lemma Component_ok_iff:
  "(Component i ok G) =
  (G  $\in$  preserves (%s. c s i) & Component i  $\in$  Allowed G)"
  <proof>
declare Component_ok_iff [iff]
declare OK_iff_ok [iff]
declare preserves_def [simp]

lemma p2: "Component i  $\in$  stable {s. C s = (c s) i + k}"
  <proof>

```

```

lemma p3:
  "[| OK I Component; i∈I |]
  ==> Component i ∈ stable {s. ∀j∈I. j≠i --> c s j = c k j}"
⟨proof⟩

lemma p2_p3_lemma1:
  "[| OK {i. i<I} Component; i<I |] ==>
  ∀k. Component i ∈ stable ({s. C s = c s i + sumj I i k} Int
  {s. ∀j∈{i. i<I}. j≠i --> c s j = c k j})"
⟨proof⟩

lemma p2_p3_lemma2:
  "(∀k. F ∈ stable ({s. C s = (c s) i + sumj I i k} Int
  {s. ∀j∈{i. i<I}. j≠i --> c s j = c k j}))
  ==> (F ∈ stable {s. C s = c s i + sumj I i s})"
⟨proof⟩

lemma p2_p3:
  "[| OK {i. i<I} Component; i<I |]
  ==> Component i ∈ stable {s. C s = c s i + sumj I i s}"
⟨proof⟩

lemma safety:
  "[| 0<I; OK {i. i<I} Component |]
  ==> (⋂ i∈{i. i<I}. (Component i)) ∈ invariant {s. C s = sum I s}"
⟨proof⟩

end

theory PriorityAux
imports "../UNITY_Main"
begin

typedecl vertex

definition symcl :: "(vertex*vertex)set=>(vertex*vertex)set" where
  "symcl r == r ∪ (r-1)"
  — symmetric closure: removes the orientation of a relation

definition neighbors :: "[vertex, (vertex*vertex)set]=>vertex set" where
  "neighbors i r == ((r ∪ r-1)''{i} - {i})"
  — Neighbors of a vertex i

definition R :: "[vertex, (vertex*vertex)set]=>vertex set" where
  "R i r == r''{i}"

definition A :: "[vertex, (vertex*vertex)set]=>vertex set" where
  "A i r == (r-1)''{i}"

```

definition *reach* :: "[vertex, (vertex*vertex)set]=> vertex set" where
 "reach i r == (r⁺)' '{i}"
 — reachable and above vertices: the original notation was R* and A*

definition *above* :: "[vertex, (vertex*vertex)set]=> vertex set" where
 "above i r == ((r⁻¹)⁺)' '{i}"

definition *reverse* :: "[vertex, (vertex*vertex) set]=>(vertex*vertex)set" where
 "reverse i r == (r - {(x,y). x=i | y=i} ∩ r) ∪ {(x,y). x=i/y=i} ∩ r)⁻¹"

definition *derive1* :: "[vertex, (vertex*vertex)set, (vertex*vertex)set]=>bool"
 where

— The original definition
 "derive1 i r q == symcl r = symcl q &
 (∀ k k'. k ≠ i & k' ≠ i --> ((k,k') ∈ r) = ((k,k') ∈ q)) ∧
 A i r = {} & R i q = {}"

definition *derive* :: "[vertex, (vertex*vertex)set, (vertex*vertex)set]=>bool"
 where

— Our alternative definition
 "derive i r q == A i r = {} & (q = reverse i r)"

axiomatization where

finite_vertex_univ: "finite (UNIV :: vertex set)"
 — we assume that the universe of vertices is finite

declare *derive_def* [simp] *derive1_def* [simp] *symcl_def* [simp]
A_def [simp] *R_def* [simp]
above_def [simp] *reach_def* [simp]
reverse_def [simp] *neighbors_def* [simp]

All vertex sets are finite

declare *finite_subset* [OF subset_UNIV finite_vertex_univ, iff]

and relations over vertex are finite too

lemmas *finite_UNIV_Prod* =
finite_Prod_UNIV [OF finite_vertex_univ finite_vertex_univ]

declare *finite_subset* [OF subset_UNIV finite_UNIV_Prod, iff]

lemma *image0_trancl_iff_image0_r*: "((r⁺)' '{i} = {}) = (r' '{i} = {})"
 <proof>

lemma *image0_r_iff_image0_trancl*: "(r' '{i}={}) = (∀ x. ((i,x) ∈ r⁺) = False)"
 <proof>

lemma *acyclic_eq_wf*: "!!r::(vertex*vertex)set. acyclic r = wf r"
 <proof>

lemma *derive_derive1_eq*: "derive i r q = derive1 i r q"
 <proof>

lemma *lemma1_a*:
 "[| x ∈ reach i q; derive1 k r q |] ==> x≠k --> x ∈ reach i r"
 <proof>

lemma *reach_lemma*: "derive k r q ==> reach i q ⊆ (reach i r ∪ {k})"
 <proof>

lemma *reach_above_lemma*:
 "(∀i. reach i q ⊆ (reach i r ∪ {k})) =
 (∀x. x≠k --> (∀i. i ∉ above x r --> i ∉ above x q))"
 <proof>

lemma *maximal_converse_image0*:
 "(z, i) ∈ r⁺ ==> (∀y. (y, z) ∈ r → (y, i) ∉ r⁺) = ((r⁻¹)⁺ "{z}={}")"
 <proof>

lemma *above_lemma_a*:
 "acyclic r ==> A i r≠{}-->(∃j ∈ above i r. A j r = {})"
 <proof>

lemma *above_lemma_b*:
 "acyclic r ==> above i r≠{}-->(∃j ∈ above i r. above j r = {})"
 <proof>

end

22 The priority system

theory *Priority imports PriorityAux begin*

From Charpentier and Chandy, Examples of Program Composition Illustrating the Use of Universal Properties In J. Rolim (editor), Parallel and Distributed Processing, Spriner LNCS 1586 (1999), pages 1215-1227.

type_synonym *state* = "(vertex*vertex)set"
type_synonym *command* = "vertex=>(state*state)set"

consts
init :: "(vertex*vertex)set"
 — the initial state

Following the definitions given in section 4.4

definition *highest* :: "[vertex, (vertex*vertex)set]=>bool"
where "*highest i r* ↔ A i r = {}"
 — i has highest priority in r

```

definition lowest :: "[vertex, (vertex*vertex)set]=>bool"
  where "lowest i r  $\longleftrightarrow$  R i r = {}"
    — i has lowest priority in r

definition act :: command
  where "act i = {(s, s'). s'=reverse i s & highest i s}"

definition Component :: "vertex=>state program"
  where "Component i = mk_total_program({init}, {act i}, UNIV)"
    — All components start with the same initial state

```

Some Abbreviations

```

definition Highest :: "vertex=>state set"
  where "Highest i = {s. highest i s}"

definition Lowest :: "vertex=>state set"
  where "Lowest i = {s. lowest i s}"

definition Acyclic :: "state set"
  where "Acyclic = {s. acyclic s}"

definition Maximal :: "state set"
  — Every "above" set has a maximal vertex
  where "Maximal = ( $\bigcap$  i. {s. ~highest i s  $\rightarrow$  ( $\exists$  j  $\in$  above i s. highest j s)})"

definition Maximal' :: "state set"
  — Maximal vertex: equivalent definition
  where "Maximal' = ( $\bigcap$  i. Highest i Un ( $\bigcup$  j. {s. j  $\in$  above i s} Int Highest j))"

definition Safety :: "state set"
  where "Safety = ( $\bigcap$  i. {s. highest i s  $\rightarrow$  ( $\forall$  j  $\in$  neighbors i s. ~highest j s)})"

```

```

definition system :: "state program"
  where "system = ( $\bigcup$  i. Component i)"

```

```

declare highest_def [simp] lowest_def [simp]
declare Highest_def [THEN def_set_simp, simp]
  and Lowest_def [THEN def_set_simp, simp]

declare Component_def [THEN def_prg_Init, simp]
declare act_def [THEN def_act_simp, simp]

```

22.1 Component correctness proofs

neighbors is stable

p15: universal property: all Components well behave

lemma *system_well_behaves*: "system \in Highest i co Highest i Un Lowest i"
 ⟨proof⟩

lemma *Acyclic_eq*: "Acyclic = $(\bigcap i. \{s. i \notin \text{above } i \ s\})$ "
 ⟨proof⟩

lemmas *system_co* =
 constrains_Un [OF above_not_increase [rule_format] system_well_behaves]

lemma *Acyclic_stable*: "system \in stable Acyclic"
 ⟨proof⟩

lemma *Acyclic_subset_Maximal*: "Acyclic \leq Maximal"
 ⟨proof⟩

property 17: original one is an invariant

lemma *Acyclic_Maximal_stable*: "system \in stable (Acyclic Int Maximal)"
 ⟨proof⟩

property 5: existential property

lemma *Highest_leadsTo_Lowest*: "system \in Highest i leadsTo Lowest i"
 ⟨proof⟩

a lowest i can never be in any above set

lemma *Lowest_above_subset*: "Lowest i \leq $(\bigcap k. \{s. i \notin \text{above } k \ s\})$ "
 ⟨proof⟩

property 18: a simpler proof than the original, one which uses psp

lemma *Highest_escapes_above*: "system \in Highest i leadsTo $(\bigcap k. \{s. i \notin \text{above } k \ s\})$ "
 ⟨proof⟩

lemma *Highest_escapes_above'*:
 "system \in Highest j Int $\{s. j \in \text{above } i \ s\}$ leadsTo $\{s. j \notin \text{above } i \ s\}$ "
 ⟨proof⟩

22.3 The main result: above set decreases

The original proof of the following formula was wrong

lemma *Highest_iff_above0*: "Highest i = $\{s. \text{above } i \ s = \{\}\}$ "
 ⟨proof⟩

lemmas *above_decreases_lemma* =
 psp [THEN leadsTo_weaken, OF Highest_escapes_above' above_not_increase']

```

lemma above_decreases:
  "system  $\in$  ( $\bigcup j$ . {s. above i s = x} Int {s. j  $\in$  above i s} Int Highest
j)
  leadsTo {s. above i s < x}"
<proof>

lemma Maximal_eq_Maximal': "Maximal = Maximal'"
<proof>

lemma Acyclic_subset:
  "x  $\neq$  {} ==>
  Acyclic Int {s. above i s = x} <=
  ( $\bigcup j$ . {s. above i s = x} Int {s. j  $\in$  above i s} Int Highest j)"
<proof>

lemmas above_decreases' = leadsTo_weaken_L [OF above_decreases Acyclic_subset]
lemmas above_decreases_psp = psp_stable [OF above_decreases' Acyclic_stable]

lemma above_decreases_psp':
  "x  $\neq$  {} ==> system  $\in$  Acyclic Int {s. above i s = x} leadsTo
  Acyclic Int {s. above i s < x}"
<proof>

lemmas finite_psubset_induct = wf_finite_psubset [THEN leadsTo_wf_induct]

lemma Progress: "system  $\in$  Acyclic leadsTo Highest i"
<proof>

We have proved all (relevant) theorems given in the paper. We didn't assume
any thing about the relation r. It is not necessary that r be a priority relation
as assumed in the original proof. It suffices that we start from a state which is
finite and acyclic.

end

theory TimerArray imports "../UNITY_Main" begin

type_synonym 'a state = "nat * 'a"

definition count :: "'a state => nat"
  where "count s = fst s"

definition decr :: "('a state * 'a state) set"
  where "decr = (UN n uu. {(Suc n, uu), (n,uu)})"

definition Timer :: "'a state program"
  where "Timer = mk_total_program (UNIV, {decr}, UNIV)"

declare Timer_def [THEN def_prg_Init, simp]

```

```
declare count_def [simp] decr_def [simp]
```

```
lemma Timer_leadsTo_zero: "Timer ∈ UNIV leadsTo {s. count s = 0}"
⟨proof⟩
```

```
lemma Timer_preserves_snd [iff]: "Timer ∈ preserves_snd"
⟨proof⟩
```

```
declare PLam_stable [simp]
```

```
lemma TimerArray_leadsTo_zero:
  "finite I
   ⇒ (plam i: I. Timer) ∈ UNIV leadsTo {(s,uu). ∀i∈I. s i = 0}"
⟨proof⟩
```

```
end
```

23 Progress Set Examples

```
theory Progress imports "../UNITY_Main" begin
```

23.1 The Composition of Two Single-Assignment Programs

Thesis Section 4.4.2

```
definition FF :: "int program" where
  "FF = mk_total_program (UNIV, {range (λx. (x, x+1))}, UNIV)"
```

```
definition GG :: "int program" where
  "GG = mk_total_program (UNIV, {range (λx. (x, 2*x))}, UNIV)"
```

23.1.1 Calculating wens_set FF {k..}

```
lemma Domain_actFF: "Domain (range (λx::int. (x, x + 1))) = UNIV"
⟨proof⟩
```

```
lemma FF_eq:
  "FF = mk_program (UNIV, {range (λx. (x, x+1))}, UNIV)"
⟨proof⟩
```

```
lemma wp_actFF:
  "wp (range (λx::int. (x, x + 1))) (atLeast k) = atLeast (k - 1)"
⟨proof⟩
```

```
lemma wens_FF: "wens FF (range (λx. (x, x+1))) (atLeast k) = atLeast (k - 1)"
⟨proof⟩
```

```
lemma single_valued_actFF: "single_valued (range (λx::int. (x, x + 1)))"
⟨proof⟩
```

```
lemma wens_single_finite_FF:
```

```

    "wens_single_finite (range ( $\lambda x. (x, x+1)$ )) (atLeast k) n =
      atLeast (k - int n)"
  <proof>

```

```

lemma wens_single_FF_eq_UNIV:
  "wens_single (range ( $\lambda x::\text{int}. (x, x + 1)$ )) (atLeast k) = UNIV"
  <proof>

```

```

lemma wens_set_FF:
  "wens_set FF (atLeast k) = insert UNIV (atLeast ' atMost k)"
  <proof>

```

23.1.2 Proving $FF \in UNIV \mapsto \{k..\}$

```

lemma atLeast_ensures: "FF  $\in$  atLeast (k - 1) ensures atLeast (k::int)"
  <proof>

```

```

lemma atLeast_leadsTo: "FF  $\in$  atLeast (k - int n) leadsTo atLeast (k::int)"
  <proof>

```

```

lemma UN_atLeast_UNIV: " $\bigcup n. \text{atLeast } (k - \text{int } n) = UNIV$ "
  <proof>

```

```

lemma FF_leadsTo: "FF  $\in$  UNIV leadsTo atLeast (k::int)"
  <proof>

```

Result (4.39): Applying the leadsTo-Join Theorem

```

theorem "FF  $\sqcup$  GG  $\in$  atLeast 0 leadsTo atLeast (k::int)"
  <proof>

```

end

24 Common Declarations for Chandy and Charpentier's Allocator

```

theory AllocBase imports "../UNITY_Main" "HOL-Library.Multiset_Order" begin

```

```

consts Nclients :: nat

```

```

axiomatization NbT :: nat
  where NbT_pos: "0 < NbT"

```

```

abbreviation (input) tokens :: "nat list  $\Rightarrow$  nat"
where

```

```

  "tokens  $\equiv$  sum_list"

```

```

abbreviation (input)
  "bag_of  $\equiv$  mset"

```

```

lemma sum_fun_mono:
  fixes f :: "nat  $\Rightarrow$  nat"
  shows " $\bigwedge i. i < n \implies f\ i \leq g\ i \implies \text{sum } f\ \{..\langle n \rangle\} \leq \text{sum } g\ \{..\langle n \rangle\}$ "

```

<proof>

lemma *tokens_mono_prefix*: "xs ≤ ys ⇒ tokens xs ≤ tokens ys"

<proof>

lemma *mono_tokens*: "mono tokens"

<proof>

lemma *bag_of_append [simp]*: "bag_of (l@l') = bag_of l + bag_of l'"

<proof>

lemma *mono_bag_of*: "mono (bag_of :: 'a list => ('a::order) multiset)"

<proof>

declare *sum.cong [cong]*

lemma *bag_of_nthn_lemma*:

" $(\sum_{i \in A} \text{Int lessThan } k. \{\# \text{if } i < k \text{ then } f \ i \ \text{else } g \ i \ \}) =$
 $(\sum_{i \in A} \text{Int lessThan } k. \{\# f \ i \ \})$ "

<proof>

lemma *bag_of_nthn*:

"bag_of (nthn l A) =
 $(\sum_{i \in A} \text{Int lessThan } (\text{length } l). \{\# l!i \ \})$ "

<proof>

lemma *bag_of_nthn_Un_Int*:

"bag_of (nthn l (A Un B)) + bag_of (nthn l (A Int B)) =
 bag_of (nthn l A) + bag_of (nthn l B)"

<proof>

lemma *bag_of_nthn_Un_disjoint*:

"A Int B = {}
 ==> bag_of (nthn l (A Un B)) =
 bag_of (nthn l A) + bag_of (nthn l B)"

<proof>

lemma *bag_of_nthn_UN_disjoint [rule_format]*:

"[| finite I; ∀ i ∈ I. ∀ j ∈ I. i ≠ j → A i Int A j = {} |]
 ==> bag_of (nthn l (⋃ (A ' I))) =
 $(\sum_{i \in I}. \text{bag_of } (\text{nthn } l \ (A \ i)))$ "

<proof>

end

theory *Alloc*

imports *AllocBase* "../PPROD"

begin

24.1 State definitions. OUTPUT variables are locals

```
record clientState =
  giv :: "nat list" — client's INPUT history: tokens GRANTED
  ask  :: "nat list" — client's OUTPUT history: tokens REQUESTED
  rel  :: "nat list" — client's OUTPUT history: tokens RELEASED
```

```
record 'a clientState_d =
  clientState +
  dummy :: 'a — dummy field for new variables
```

definition

— DUPLICATED FROM Client.thy, but with "tok" removed
 — Maybe want a special theory section to declare such maps

```
non_dummy :: "'a clientState_d => clientState"
where "non_dummy s = (|giv = giv s, ask = ask s, rel = rel s|)"
```

definition

— Renaming map to put a Client into the standard form

```
client_map :: "'a clientState_d => clientState*'a"
where "client_map = funPair non_dummy dummy"
```

```
record allocState =
  allocGiv :: "nat => nat list" — OUTPUT history: source of "giv" for i
  allocAsk :: "nat => nat list" — INPUT: allocator's copy of "ask" for i
  allocRel :: "nat => nat list" — INPUT: allocator's copy of "rel" for i
```

```
record 'a allocState_d =
  allocState +
  dummy :: 'a — dummy field for new variables
```

```
record 'a systemState =
  allocState +
  client :: "nat => clientState" — states of all clients
  dummy  :: 'a — dummy field for new variables
```

24.1.1 Resource allocation system specification

definition

— spec (1)

```
system_safety :: "'a systemState program set"
where "system_safety =
  Always {s. ( $\sum i \in \text{lessThan } N\text{clients. (tokens o giv o sub } i \text{ o client)}s$ )
   $\leq NbT + (\sum i \in \text{lessThan } N\text{clients. (tokens o rel o sub } i \text{ o client)}s)$ }"
```

definition

— spec (2)

```
system_progress :: "'a systemState program set"
where "system_progress = (INT i : lessThan Nclients.
  INT h.
  {s. h  $\leq$  (ask o sub i o client)s} LeadsTo
```

```
{s. h pfixLe (giv o sub i o client) s})"
```

definition

```
system_spec :: "'a systemState program set"
where "system_spec = system_safety Int system_progress"
```

24.1.2 Client specification (required)

definition

```
— spec (3)
client_increasing :: "'a clientState_d program set"
where "client_increasing = UNIV guarantees Increasing ask Int Increasing
rel"
```

definition

```
— spec (4)
client_bounded :: "'a clientState_d program set"
where "client_bounded = UNIV guarantees Always {s.  $\forall elt \in set (ask s).$ 
elt  $\leq$  NbT}"
```

definition

```
— spec (5)
client_progress :: "'a clientState_d program set"
where "client_progress =
Increasing giv guarantees
(INT h. {s. h  $\leq$  giv s & h pfixGe ask s}
LeadsTo {s. tokens h  $\leq$  (tokens o rel) s})"
```

definition

```
— spec: preserves part
client_preserves :: "'a clientState_d program set"
where "client_preserves = preserves giv Int preserves clientState_d.dummy"
```

definition

```
— environmental constraints
client_allowed_acts :: "'a clientState_d program set"
where "client_allowed_acts =
{F. AllowedActs F =
insert Id ( $\bigcup$  (Acts 'preserves (funPair rel ask)))}"
```

definition

```
client_spec :: "'a clientState_d program set"
where "client_spec = client_increasing Int client_bounded Int client_progress
Int client_allowed_acts Int client_preserves"
```

24.1.3 Allocator specification (required)

definition

```
— spec (6)
alloc_increasing :: "'a allocState_d program set"
where "alloc_increasing =
UNIV guarantees
(INT i : lessThan Nclients. Increasing (sub i o allocGiv))"
```

definition


```

— spec (7)
alloc_safety :: "'a allocState_d program set"
where "alloc_safety =
      (INT i : lessThan Nclients. Increasing (sub i o allocRel))
      guarantees
      Always {s. ( $\sum$  i  $\in$  lessThan Nclients. (tokens o sub i o allocGiv)s)
       $\leq$  NbT + ( $\sum$  i  $\in$  lessThan Nclients. (tokens o sub i o allocRel)s)}"
```

definition

```

— spec (8)
alloc_progress :: "'a allocState_d program set"
where "alloc_progress =
      (INT i : lessThan Nclients. Increasing (sub i o allocAsk) Int
      Increasing (sub i o allocRel))
      Int
      Always {s.  $\forall$  i < Nclients.
       $\forall$  elt  $\in$  set ((sub i o allocAsk) s). elt  $\leq$  NbT}
      Int
      (INT i : lessThan Nclients.
      INT h. {s. h  $\leq$  (sub i o allocGiv)s & h pfixGe (sub i o allocAsk)s}
      LeadsTo
      {s. tokens h  $\leq$  (tokens o sub i o allocRel)s})
      guarantees
      (INT i : lessThan Nclients.
      INT h. {s. h  $\leq$  (sub i o allocAsk) s}
      LeadsTo
      {s. h pfixLe (sub i o allocGiv) s})"
```

definition

```

— spec: preserves part
alloc_preserves :: "'a allocState_d program set"
where "alloc_preserves = preserves allocRel Int preserves allocAsk Int
      preserves allocState_d.dummy"
```

definition

```

— environmental constraints
alloc_allowed_acts :: "'a allocState_d program set"
where "alloc_allowed_acts =
      {F. AllowedActs F =
      insert Id ( $\bigcup$  (Acts ' (preserves allocGiv)))}"
```

definition

```

alloc_spec :: "'a allocState_d program set"
where "alloc_spec = alloc_increasing Int alloc_safety Int alloc_progress
      Int
      alloc_allowed_acts Int alloc_preserves"
```

24.1.4 Network specification**definition**

```

— spec (9.1)
network_ask :: "'a systemState program set"
```

```

where "network_ask = (INT i : lessThan Nclients.
    Increasing (ask o sub i o client) guarantees
    ((sub i o allocAsk) Fols (ask o sub i o client)))"

```

definition

```

— spec (9.2)
network_giv :: "'a systemState program set"
where "network_giv = (INT i : lessThan Nclients.
    Increasing (sub i o allocGiv)
    guarantees
    ((giv o sub i o client) Fols (sub i o allocGiv)))"

```

definition

```

— spec (9.3)
network_rel :: "'a systemState program set"
where "network_rel = (INT i : lessThan Nclients.
    Increasing (rel o sub i o client)
    guarantees
    ((sub i o allocRel) Fols (rel o sub i o client)))"

```

definition

```

— spec: preserves part
network_preserves :: "'a systemState program set"
where "network_preserves =
    preserves allocGiv Int
    (INT i : lessThan Nclients. preserves (rel o sub i o client) Int
    preserves (ask o sub i o client))"

```

definition

```

— environmental constraints
network_allowed_acts :: "'a systemState program set"
where "network_allowed_acts =
    {F. AllowedActs F = insert Id
    (⋃ (Acts ' (preserves allocRel ∩ (⋂ i<Nclients.
    preserves (giv o sub i o client))))}"

```

definition

```

network_spec :: "'a systemState program set"
where "network_spec = network_ask Int network_giv Int
    network_rel Int network_allowed_acts Int
    network_preserves"

```

24.1.5 State mappings**definition**

```

sysOfAlloc :: "(nat => clientState) * 'a) allocState_d => 'a systemState"
where "sysOfAlloc = (%s. let (cl,xtr) = allocState_d.dummy s
    in (| allocGiv = allocGiv s,
    allocAsk = allocAsk s,
    allocRel = allocRel s,
    client = cl,
    dummy = xtr|))"

```

definition

```

sysOfClient :: "(nat => clientState) * 'a allocState_d => 'a systemState"
where "sysOfClient = (%(cl,al). (! allocGiv = allocGiv al,
                                allocAsk = allocAsk al,
                                allocRel = allocRel al,
                                client    = cl,
                                systemState.dummy = allocState_d.dummy al)))"

```

```

axiomatization Alloc :: "'a allocState_d program"
where Alloc: "Alloc ∈ alloc_spec"

```

```

axiomatization Client :: "'a clientState_d program"
where Client: "Client ∈ client_spec"

```

```

axiomatization Network :: "'a systemState program"
where Network: "Network ∈ network_spec"

```

```

definition System :: "'a systemState program"
where "System = rename sysOfAlloc Alloc ⊔ Network ⊔
        (rename sysOfClient
         (plam x: lessThan Nclients. rename client_map Client))"

```

```

declare subset_preserves_o [THEN [2] rev_subsetD, intro]
declare subset_preserves_o [THEN [2] rev_subsetD, simp]
declare funPair_o_distrib [simp]
declare Always_INT_distrib [simp]
declare o_apply [simp del]

```

```

lemmas [simp] =
  rename_image_constrains
  rename_image_stable
  rename_image_increasing
  rename_image_invariant
  rename_image_Constrains
  rename_image_Stable
  rename_image_Increasing
  rename_image_Always
  rename_image_leadsTo
  rename_image_LeadsTo
  rename_preserves
  rename_image_preserves
  lift_image_preserves
  bij_image_INT
  bij_is_inj [THEN image_Int]
  bij_image_Collect_eq

```

⟨ML⟩

```

lemmas lessThanBspec = lessThan_iff [THEN iffD2, THEN [2] bspec]

```

<ML>

lemma *inj_sysOfAlloc [iff]: "inj sysOfAlloc"*
<proof>

We need the inverse; also having it simplifies the proof of surjectivity

lemma *inv_sysOfAlloc_eq [simp]: "!!s. inv sysOfAlloc s =*
(| allocGiv = allocGiv s,
allocAsk = allocAsk s,
allocRel = allocRel s,
allocState_d.dummy = (client s, dummy s) |)"
<proof>

lemma *surj_sysOfAlloc [iff]: "surj sysOfAlloc"*
<proof>

lemma *bij_sysOfAlloc [iff]: "bij sysOfAlloc"*
<proof>

24.1.6 bijectivity of *sysOfClient*

lemma *inj_sysOfClient [iff]: "inj sysOfClient"*
<proof>

lemma *inv_sysOfClient_eq [simp]: "!!s. inv sysOfClient s =*
(client s,
(| allocGiv = allocGiv s,
allocAsk = allocAsk s,
allocRel = allocRel s,
allocState_d.dummy = systemState.dummy s|))"
<proof>

lemma *surj_sysOfClient [iff]: "surj sysOfClient"*
<proof>

lemma *bij_sysOfClient [iff]: "bij sysOfClient"*
<proof>

24.1.7 bijectivity of *client_map*

lemma *inj_client_map [iff]: "inj client_map"*
<proof>

lemma *inv_client_map_eq [simp]: "!!s. inv client_map s =*
(%(x,y).(|giv = giv x, ask = ask x, rel = rel x,
clientState_d.dummy = y|)) s"
<proof>

lemma *surj_client_map [iff]: "surj client_map"*
<proof>

lemma *bij_client_map [iff]: "bij client_map"*
<proof>

o-simprules for *client_map*

```
lemma fst_o_client_map: "fst o client_map = non_dummy"
  <proof>
```

<ML>

```
declare fst_o_client_map' [simp]
```

```
lemma snd_o_client_map: "snd o client_map = clientState_d.dummy"
  <proof>
```

<ML>

```
declare snd_o_client_map' [simp]
```

24.2 o-simprules for *sysOfAlloc* [MUST BE AUTOMATED]

```
lemma client_o_sysOfAlloc: "client o sysOfAlloc = fst o allocState_d.dummy"
  "
  <proof>
```

<ML>

```
declare client_o_sysOfAlloc' [simp]
```

```
lemma allocGiv_o_sysOfAlloc_eq: "allocGiv o sysOfAlloc = allocGiv"
  <proof>
```

<ML>

```
declare allocGiv_o_sysOfAlloc_eq' [simp]
```

```
lemma allocAsk_o_sysOfAlloc_eq: "allocAsk o sysOfAlloc = allocAsk"
  <proof>
```

<ML>

```
declare allocAsk_o_sysOfAlloc_eq' [simp]
```

```
lemma allocRel_o_sysOfAlloc_eq: "allocRel o sysOfAlloc = allocRel"
  <proof>
```

<ML>

```
declare allocRel_o_sysOfAlloc_eq' [simp]
```

24.3 o-simprules for *sysOfClient* [MUST BE AUTOMATED]

```
lemma client_o_sysOfClient: "client o sysOfClient = fst"
  <proof>
```

<ML>

```
declare client_o_sysOfClient' [simp]
```

```
lemma allocGiv_o_sysOfClient_eq: "allocGiv o sysOfClient = allocGiv o snd"
  "
  <proof>
```

<ML>

```
declare allocGiv_o_sysOfClient_eq' [simp]
```

15024 COMMON DECLARATIONS FOR CHANDY AND CHARPENTIER'S ALLOCATOR

```

lemma allocAsk_o_sysOfClient_eq: "allocAsk o sysOfClient = allocAsk o snd
"
  <proof>

  <ML>
declare allocAsk_o_sysOfClient_eq' [simp]

lemma allocRel_o_sysOfClient_eq: "allocRel o sysOfClient = allocRel o snd
"
  <proof>

  <ML>
declare allocRel_o_sysOfClient_eq' [simp]

lemma allocGiv_o_inv_sysOfAlloc_eq: "allocGiv o inv sysOfAlloc = allocGiv"
  <proof>

  <ML>
declare allocGiv_o_inv_sysOfAlloc_eq' [simp]

lemma allocAsk_o_inv_sysOfAlloc_eq: "allocAsk o inv sysOfAlloc = allocAsk"
  <proof>

  <ML>
declare allocAsk_o_inv_sysOfAlloc_eq' [simp]

lemma allocRel_o_inv_sysOfAlloc_eq: "allocRel o inv sysOfAlloc = allocRel"
  <proof>

  <ML>
declare allocRel_o_inv_sysOfAlloc_eq' [simp]

lemma rel_inv_client_map_drop_map: "(rel o inv client_map o drop_map i o
inv sysOfClient) =
  rel o sub i o client"
  <proof>

  <ML>
declare rel_inv_client_map_drop_map [simp]

lemma ask_inv_client_map_drop_map: "(ask o inv client_map o drop_map i o
inv sysOfClient) =
  ask o sub i o client"
  <proof>

  <ML>
declare ask_inv_client_map_drop_map [simp]

Client : <unfolded specification>

lemmas client_spec_simps =
  client_spec_def client_increasing_def client_bounded_def
  client_progress_def client_allowed_acts_def client_preserves_def
  guarantees_Int_right

```

<ML>

declare

```
Client_Increasing_ask [iff]
Client_Increasing_rel [iff]
Client_Bounded [iff]
Client_preserves_giv [iff]
Client_preserves_dummy [iff]
```

Network : <unfolded specification>

lemmas network_spec_simps =

```
network_spec_def network_ask_def network_giv_def
network_rel_def network_allowed_acts_def network_preserves_def
ball_conj_distrib
```

<ML>

declare Network_preserves_allocGiv [iff]

declare

```
Network_preserves_rel [simp]
Network_preserves_ask [simp]
```

declare

```
Network_preserves_rel [simplified o_def, simp]
Network_preserves_ask [simplified o_def, simp]
```

Alloc : <unfolded specification>

lemmas alloc_spec_simps =

```
alloc_spec_def alloc_increasing_def alloc_safety_def
alloc_progress_def alloc_allowed_acts_def alloc_preserves_def
```

<ML>

Strip off the INT in the guarantees postcondition

lemmas Alloc_Increasing = Alloc_Increasing_0 [normalized]

declare

```
Alloc_preserves_allocRel [iff]
Alloc_preserves_allocAsk [iff]
Alloc_preserves_dummy [iff]
```

24.4 Components Lemmas [MUST BE AUTOMATED]

lemma Network_component_System: "Network \sqsubseteq

```
((rename sysOfClient
  (plam x: (lessThan Nclients). rename client_map Client))  $\sqsubseteq$ 
  rename sysOfAlloc Alloc)
= System"
```

<proof>

lemma Client_component_System: "(rename sysOfClient

```
(plam x: (lessThan Nclients). rename client_map Client))  $\sqsubseteq$ 
```

```

      (Network  $\sqcup$  rename sysOfAlloc Alloc) = System"
    <proof>

lemma Alloc_component_System: "rename sysOfAlloc Alloc  $\sqcup$ 
  ((rename sysOfClient (plam x: (lessThan Nclients). rename client_map
  Client))  $\sqcup$ 
  Network) = System"
  <proof>

declare
  Client_component_System [iff]
  Network_component_System [iff]
  Alloc_component_System [iff]

* These preservation laws should be generated automatically *

lemma Client_Allowed [simp]: "Allowed Client = preserves rel Int preserves
ask"
  <proof>

lemma Network_Allowed [simp]: "Allowed Network =
  preserves allocRel Int
  (INT i: lessThan Nclients. preserves(giv o sub i o client))"
  <proof>

lemma Alloc_Allowed [simp]: "Allowed Alloc = preserves allocGiv"
  <proof>

needed in rename_client_map_tac

lemma OK_lift_rename_Client [simp]: "OK I (%i. lift i (rename client_map
Client))"
  <proof>

lemma fst_lift_map_eq_fst [simp]: "fst (lift_map i x) i = fst x"
  <proof>

lemma fst_o_lift_map' [simp]:
  "(f o sub i o fst o lift_map i o g) = f o fst o g"
  <proof>

<ML>

Lifting Client_Increasing to systemState

lemma rename_Client_Increasing: "i  $\in$  I
==> rename sysOfClient (plam x: I. rename client_map Client)  $\in$ 
  UNIV guarantees
  Increasing (ask o sub i o client) Int
  Increasing (rel o sub i o client)"
  <proof>

lemma preserves_sub_fst_lift_map: "[| F  $\in$  preserves w; i  $\neq$  j |]
==> F  $\in$  preserves (sub i o fst o lift_map j o funPair v w)"

```


<proof>

lemma *client_preserves_giv_oo_client_map*: "[| i < Nclients; j < Nclients
|]
==> Client ∈ preserves (giv o sub i o fst o lift_map j o client_map)"
<proof>

lemma *rename_sysOfClient_ok_Network*:
"rename sysOfClient (plam x: lessThan Nclients. rename client_map Client)
ok Network"
<proof>

lemma *rename_sysOfClient_ok_Alloc*:
"rename sysOfClient (plam x: lessThan Nclients. rename client_map Client)
ok rename sysOfAlloc Alloc"
<proof>

lemma *rename_sysOfAlloc_ok_Network*: "rename sysOfAlloc Alloc ok Network"
<proof>

declare
rename_sysOfClient_ok_Network [iff]
rename_sysOfClient_ok_Alloc [iff]
rename_sysOfAlloc_ok_Network [iff]

The "ok" laws, re-oriented. But not sure this works: theorem *ok_commute* is needed below

declare
rename_sysOfClient_ok_Network [THEN ok_sym, iff]
rename_sysOfClient_ok_Alloc [THEN ok_sym, iff]
rename_sysOfAlloc_ok_Network [THEN ok_sym]

lemma *System_Increasing*: "i < Nclients
==> System ∈ Increasing (ask o sub i o client) Int
Increasing (rel o sub i o client)"
<proof>

lemmas *rename_guarantees_sysOfAlloc_I =*
bij_sysOfAlloc [THEN rename_rename_guarantees_eq, THEN iffD2]

lemmas *rename_Alloc_Increasing =*
Alloc_Increasing
[THEN rename_guarantees_sysOfAlloc_I,
simplified surj_rename o_def sub_apply
rename_image_Increasing bij_sysOfAlloc
allocGiv_o_inv_sysOfAlloc_eq']

lemma *System_Increasing_allocGiv*:
"i < Nclients ==> System ∈ Increasing (sub i o allocGiv)"
<proof>

<ML>

declare *System_Increasing'* [intro!]

Follows consequences. The "Always (INT ...) formulation expresses the general safety property and allows it to be combined using *Always_Int_rule* below.

lemma *System_Follows_rel*:

"*i* < *Nclients* ==> *System* ∈ ((sub *i* o allocRel) Fols (rel o sub *i* o client))"
<proof>

lemma *System_Follows_ask*:

"*i* < *Nclients* ==> *System* ∈ ((sub *i* o allocAsk) Fols (ask o sub *i* o client))"
<proof>

lemma *System_Follows_allocGiv*:

"*i* < *Nclients* ==> *System* ∈ (giv o sub *i* o client) Fols (sub *i* o allocGiv)"
<proof>

lemma *Always_giv_le_allocGiv*: "*System* ∈ Always (INT *i*: lessThan *Nclients*.
 {*s*. (giv o sub *i* o client) *s* ≤ (sub *i* o allocGiv) *s*})"
<proof>

lemma *Always_allocAsk_le_ask*: "*System* ∈ Always (INT *i*: lessThan *Nclients*.
 {*s*. (sub *i* o allocAsk) *s* ≤ (ask o sub *i* o client) *s*})"
<proof>

lemma *Always_allocRel_le_rel*: "*System* ∈ Always (INT *i*: lessThan *Nclients*.
 {*s*. (sub *i* o allocRel) *s* ≤ (rel o sub *i* o client) *s*})"
<proof>

24.5 Proof of the safety property (1)

safety (1), step 1 is *System_Follows_rel*

safety (1), step 2

lemmas *System_Increasing_allocRel* = *System_Follows_rel* [THEN *Follows_Increasing1*]

safety (1), step 3

lemma *System_sum_bounded*:

"*System* ∈ Always {*s*. (\sum *i* ∈ lessThan *Nclients*. (tokens o sub *i* o allocGiv)
s)
 ≤ *NbT* + (\sum *i* ∈ lessThan *Nclients*. (tokens o sub *i* o allocRel)
s)}"
<proof>

Follows reasoning

lemma *Always_tokens_giv_le_allocGiv*: "*System* ∈ Always (INT *i*: lessThan *Nclients*.
 {*s*. (tokens o giv o sub *i* o client) *s*
 ≤ (tokens o sub *i* o allocGiv) *s*})"
<proof>

lemma *Always_tokens_allocRel_le_rel*: "System \in Always (INT i: lessThan Nclients.
 {s. (tokens o sub i o allocRel) s
 \leq (tokens o rel o sub i o client) s})"

<proof>

safety (1), step 4 (final result!)

theorem *System_safety*: "System \in system_safety"

<proof>

24.6 Proof of the progress property (2)

progress (2), step 1 is *System_Follows_ask* and *System_Follows_rel*

progress (2), step 2; see also *System_Increasing_allocRel*

lemmas *System_Increasing_allocAsk = System_Follows_ask [THEN Follows_Increasing1]*

progress (2), step 3: lifting *Client_Bounded* to systemState

lemma *rename_Client_Bounded*: "i \in I
 \implies rename sysOfClient (plam x: I. rename client_map Client) \in
 UNIV guarantees
 Always {s. \forall elt \in set ((ask o sub i o client) s). elt \leq NbT}"

<proof>

lemma *System_Bounded_ask*: "i < Nclients
 \implies System \in Always
 {s. \forall elt \in set ((ask o sub i o client) s). elt \leq NbT}"

<proof>

lemma *Collect_all_imp_eq*: "{x. \forall y. P y \longrightarrow Q x y} = (INT y: {y. P y}. {x.
 Q x y})"

<proof>

progress (2), step 4

lemma *System_Bounded_allocAsk*: "System \in Always {s. \forall i < Nclients.
 \forall elt \in set ((sub i o allocAsk) s). elt \leq NbT}"

<proof>

progress (2), step 5 is *System_Increasing_allocGiv*

progress (2), step 6

lemmas *System_Increasing_giv = System_Follows_allocGiv [THEN Follows_Increasing1]*

lemma *rename_Client_Progress*: "i \in I
 \implies rename sysOfClient (plam x: I. rename client_map Client)
 \in Increasing (giv o sub i o client)
 guarantees
 (INT h. {s. h \leq (giv o sub i o client) s &
 h prefixGe (ask o sub i o client) s}
 LeadsTo {s. tokens h \leq (tokens o rel o sub i o client) s})"

<proof>

progress (2), step 7

```
lemma System_Client_Progress:
  "System ∈ (INT i : (lessThan Nclients).
    INT h. {s. h ≤ (giv o sub i o client) s &
      h pfixGe (ask o sub i o client) s}
    LeadsTo {s. tokens h ≤ (tokens o rel o sub i o client) s})"
  ⟨proof⟩
```

```
lemmas System_lemma1 =
  Always_LeadsToD [OF System_Follows_ask [THEN Follows_Bounded]
    System_Follows_allocGiv [THEN Follows_LeadsTo]]
```

```
lemmas System_lemma2 =
  PSP_Stable [OF System_lemma1
    System_Follows_ask [THEN Follows_Increasing1, THEN IncreasingD]]
```

```
lemma System_lemma3: "i < Nclients
  ==> System ∈ {s. h ≤ (sub i o allocGiv) s &
    h pfixGe (sub i o allocAsk) s}
  LeadsTo
  {s. h ≤ (giv o sub i o client) s &
    h pfixGe (ask o sub i o client) s}"
  ⟨proof⟩
```

progress (2), step 8: Client i's "release" action is visible system-wide

```
lemma System_Alloc_Client_Progress: "i < Nclients
  ==> System ∈ {s. h ≤ (sub i o allocGiv) s &
    h pfixGe (sub i o allocAsk) s}
  LeadsTo {s. tokens h ≤ (tokens o sub i o allocRel) s}"
  ⟨proof⟩
```

Lifting *Alloc_Progress* up to the level of *systemState*

progress (2), step 9

```
lemma System_Alloc_Progress:
  "System ∈ (INT i : (lessThan Nclients).
    INT h. {s. h ≤ (sub i o allocAsk) s}
    LeadsTo {s. h pfixLe (sub i o allocGiv) s})"
  ⟨proof⟩
```

progress (2), step 10 (final result!)

```
lemma System_Progress: "System ∈ system_progress"
  ⟨proof⟩
```

```
theorem System_correct: "System ∈ system_spec"
  ⟨proof⟩
```

Some obsolete lemmas

```
lemma non_dummy_eq_o_funPair: "non_dummy = (% (g,a,r). (| giv = g, ask =
a, rel = r |)) o
```

```
(funPair giv (funPair ask rel))"
```

```
<proof>
```

```
lemma preserves_non_dummy_eq: "(preserves non_dummy) =
(preserves rel Int preserves ask Int preserves giv)"
```

```
<proof>
```

Could go to Extend.ML

```
lemma bij_fst_inv_inv_eq: "bij f ==> fst (inv (%(x, u). inv f x) z) = f z"
```

```
<proof>
```

```
end
```

25 Implementation of a multiple-client allocator from a single-client allocator

```
theory AllocImpl imports AllocBase "../Follows" "../PPROD" begin
```

```
record 'b merge =
  In  :: "nat => 'b list"
  Out :: "'b list"
  iOut :: "nat list"
```

```
record ('a,'b) merge_d =
  "'b merge" +
  dummy :: 'a
```

```
definition non_dummy :: "('a,'b) merge_d => 'b merge" where
  "non_dummy s = (|In = In s, Out = Out s, iOut = iOut s|)"
```

```
record 'b distr =
  In  :: "'b list"
  iIn :: "nat list"
  Out :: "nat => 'b list"
```

```
record ('a,'b) distr_d =
  "'b distr" +
  dummy :: 'a
```

```
record allocState =
  giv :: "nat list"
  ask  :: "nat list"
  rel  :: "nat list"
```

```
record 'a allocState_d =
  allocState +
  dummy      :: 'a
```

```

record 'a systemState =
  allocState +
  mergeRel :: "nat merge"
  mergeAsk :: "nat merge"
  distr    :: "nat distr"
  dummy    :: 'a

```

definition

```

merge_increasing :: "('a,'b) merge_d program set"
where "merge_increasing =
      UNIV guarantees (Increasing merge.Out) Int (Increasing merge.iOut)"

```

definition

```

merge_eqOut :: "('a,'b) merge_d program set"
where "merge_eqOut =
      UNIV guarantees
      Always {s. length (merge.Out s) = length (merge.iOut s)}"

```

definition

```

merge_bounded :: "('a,'b) merge_d program set"
where "merge_bounded =
      UNIV guarantees
      Always {s.  $\forall$ elt  $\in$  set (merge.iOut s). elt < Nclients}"

```

definition

```

merge_follows :: "('a,'b) merge_d program set"
where "merge_follows =
      ( $\bigcap$  i  $\in$  lessThan Nclients. Increasing (sub i o merge.In))
      guarantees
      ( $\bigcap$  i  $\in$  lessThan Nclients.
      (%s. nth s (merge.Out s)
      {k. k < size(merge.iOut s) & merge.iOut s! k = i})
      Fols (sub i o merge.In))"

```

definition

```

merge_preserves :: "('a,'b) merge_d program set"
where "merge_preserves = preserves merge.In Int preserves merge.dummy"

```

definition

```

merge_allowed_acts :: "('a,'b) merge_d program set"
where "merge_allowed_acts =
      {F. AllowedActs F =
      insert Id ( $\bigcup$  (Acts 'preserves (funPair merge.Out iOut)))}"

```

definition

```
merge_spec ::>('a,'b) merge_d program set"
where "merge_spec = merge_increasing Int merge_eqOut Int merge_bounded Int
      merge_follows Int merge_allowed_acts Int merge_preserves"
```

definition

```
distr_follows ::>('a,'b) distr_d program set"
where "distr_follows =
      Increasing distr.In Int Increasing distr.iIn Int
      Always {s.  $\forall \text{elt} \in \text{set} (\text{distr.iIn } s). \text{elt} < \text{Nclients}$ }
      guarantees
      ( $\bigcap i \in \text{lessThan } \text{Nclients}.$ 
       (sub i o distr.Out) Fols
       (%s. nths (distr.In s)
                {k. k < size(distr.iIn s) & distr.iIn s ! k = i}))"
```

definition

```
distr_allowed_acts ::>('a,'b) distr_d program set"
where "distr_allowed_acts =
      {D. AllowedActs D = insert Id ( $\bigcup (\text{Acts ' (preserves distr.Out))$ )}"
```

definition

```
distr_spec ::>('a,'b) distr_d program set"
where "distr_spec = distr_follows Int distr_allowed_acts"
```

definition

```
alloc_increasing :: 'a allocState_d program set"
where "alloc_increasing = UNIV guarantees Increasing giv"
```

definition

```
alloc_safety :: 'a allocState_d program set"
where "alloc_safety =
      Increasing rel
      guarantees Always {s. tokens (giv s)  $\leq$  NbT + tokens (rel s)}"
```

definition

```
alloc_progress :: 'a allocState_d program set"
where "alloc_progress =
      Increasing ask Int Increasing rel Int
      Always {s.  $\forall \text{elt} \in \text{set} (\text{ask } s). \text{elt} \leq \text{NbT}$ }
      Int
      ( $\bigcap h. \{s. h \leq \text{giv } s \ \& \ h \text{ prefixGe } (\text{ask } s)\}$ 
       LeadsTo
       {s. tokens h  $\leq$  tokens (rel s)})
      guarantees ( $\bigcap h. \{s. h \leq \text{ask } s\}$  LeadsTo {s. h prefixLe giv s})"
```

definition

```
alloc_preserves :: "'a allocState_d program set"
where "alloc_preserves = preserves rel Int
      preserves ask Int
      preserves allocState_d.dummy"
```

definition

```
alloc_allowed_acts :: "'a allocState_d program set"
where "alloc_allowed_acts =
      {F. AllowedActs F = insert Id (⋃ (Acts ' (preserves giv)))}"
```

definition

```
alloc_spec :: "'a allocState_d program set"
where "alloc_spec = alloc_increasing Int alloc_safety Int alloc_progress
      Int
      alloc_allowed_acts Int alloc_preserves"
```

locale Merge =

```
fixes M :: "('a, 'b::order) merge_d program"
assumes
  Merge_spec: "M ∈ merge_spec"
```

locale Distrib =

```
fixes D :: "('a, 'b::order) distr_d program"
assumes
  Distrib_spec: "D ∈ distr_spec"
```

```
declare subset_preserves_o [THEN subsetD, intro]
declare funPair_o_distrib [simp]
declare Always_INT_distrib [simp]
declare o_apply [simp del]
```

25.1 Theorems for Merge

context Merge

begin

lemma Merge_Allowed:

```
"Allowed M = (preserves merge.Out) Int (preserves merge.iOut)"
⟨proof⟩
```

lemma M_ok_iff [iff]:

```
"M ok G = (G ∈ preserves merge.Out & G ∈ preserves merge.iOut &
           M ∈ Allowed G)"
⟨proof⟩
```



```

lemma Merge_Always_Out_eq_iOut:
  "[| G ∈ preserves merge.Out; G ∈ preserves merge.iOut; M ∈ Allowed G
  |]
  ==> M ⊔ G ∈ Always {s. length (merge.Out s) = length (merge.iOut s)}"
  <proof>

lemma Merge_Bounded:
  "[| G ∈ preserves merge.iOut; G ∈ preserves merge.Out; M ∈ Allowed G
  |]
  ==> M ⊔ G ∈ Always {s. ∀elt ∈ set (merge.iOut s). elt < Nclients}"
  <proof>

lemma Merge_Bag_Follows_lemma:
  "[| G ∈ preserves merge.iOut; G ∈ preserves merge.Out; M ∈ Allowed G
  |]
  ==> M ⊔ G ∈ Always
    {s. (∑ i ∈ lessThan Nclients. bag_of (nth (merge.Out s)
      {k. k < length (iOut s) & iOut s ! k = i}))
  =
    (bag_of o merge.Out) s}"
  <proof>

lemma Merge_Bag_Follows:
  "M ∈ (∩ i ∈ lessThan Nclients. Increasing (sub i o merge.In))
  guarantees
    (bag_of o merge.Out) Fols
    (%s. ∑ i ∈ lessThan Nclients. (bag_of o sub i o merge.In) s)"
  <proof>

end

```

25.2 Theorems for Distributor

```

context Distrib
begin

```

```

lemma Distr_Increasing_Out:
  "D ∈ Increasing distr.In Int Increasing distr.iIn Int
  Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients}
  guarantees
    (∩ i ∈ lessThan Nclients. Increasing (sub i o distr.Out))"
  <proof>

lemma Distr_Bag_Follows_lemma:
  "[| G ∈ preserves distr.Out;
  D ⊔ G ∈ Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients} |]
  ==> D ⊔ G ∈ Always
    {s. (∑ i ∈ lessThan Nclients. bag_of (nth (distr.In s)
      {k. k < length (iIn s) & iIn s ! k = i}))
  =
    bag_of (nth (distr.In s) (lessThan (length (iIn s))))}"
  <proof>

lemma D_ok_iff [iff]:

```

```

    "D ok G = (G ∈ preserves distr.Out & D ∈ Allowed G)"
  ⟨proof⟩

lemma Distr_Bag_Follows:
  "D ∈ Increasing distr.In Int Increasing distr.iIn Int
   Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients}
   guarantees
   (∩ i ∈ lessThan Nclients.
    (%s. ∑ i ∈ lessThan Nclients. (bag_of o sub i o distr.Out) s)
    Fols
    (%s. bag_of (nth (distr.In s) (lessThan (length(distr.iIn s))))))"
  ⟨proof⟩

end

```

25.3 Theorems for Allocator

```

lemma alloc_refinement_lemma:
  "!!f::nat=>nat. (∩ i ∈ lessThan n. {s. f i ≤ g i s})
   ⊆ {s. (∑ x ∈ lessThan n. f x) ≤ (∑ x ∈ lessThan n. g x s)}"
  ⟨proof⟩

lemma alloc_refinement:
  "(∩ i ∈ lessThan Nclients. Increasing (sub i o allocAsk) Int
   Increasing (sub i o allocRel))
   Int
   Always {s. ∀i. i < Nclients -->
    (∀elt ∈ set ((sub i o allocAsk) s). elt ≤ NbT)}
   Int
   (∩ i ∈ lessThan Nclients.
    ∩ h. {s. h ≤ (sub i o allocGiv)s & h prefixGe (sub i o allocAsk)s}
    LeadsTo {s. tokens h ≤ (tokens o sub i o allocRel)s})
   ⊆
   (∩ i ∈ lessThan Nclients. Increasing (sub i o allocAsk) Int
   Increasing (sub i o allocRel))
   Int
   Always {s. ∀i. i < Nclients -->
    (∀elt ∈ set ((sub i o allocAsk) s). elt ≤ NbT)}
   Int
   (∩ hf. (∩ i ∈ lessThan Nclients.
    {s. hf i ≤ (sub i o allocGiv)s & hf i prefixGe (sub i o allocAsk)s}
    LeadsTo {s. (∑ i ∈ lessThan Nclients. tokens (hf i)) ≤
    (∑ i ∈ lessThan Nclients. (tokens o sub i o allocRel)s}))"
  ⟨proof⟩

end

```

26 Distributed Resource Management System: the Client

```
theory Client imports "../Rename" AllocBase begin
```

```
type_synonym
```

```

tokbag = nat      — tokbags could be multisets...or any ordered type?

record state =
  giv :: "tokbag list" — input history: tokens granted
  ask  :: "tokbag list" — output history: tokens requested
  rel  :: "tokbag list" — output history: tokens released
  tok  :: tokbag       — current token request

record 'a state_d =
  state +
  dummy :: 'a      — new variables

definition
rel_act :: "('a state_d * 'a state_d) set"
where "rel_act = {(s,s').
      ∃ nrel. nrel = size (rel s) &
      s' = s (| rel := rel s @ [giv s!nrel] |) &
      nrel < size (giv s) &
      ask s!nrel ≤ giv s!nrel}"

definition
tok_act :: "('a state_d * 'a state_d) set"
where "tok_act = {(s,s'). s'=s | s' = s (|tok := Suc (tok s mod NbT) |)}"

definition
ask_act :: "('a state_d * 'a state_d) set"
where "ask_act = {(s,s'). s'=s |
      (s' = s (|ask := ask s @ [tok s]|))}"

definition
Client :: "'a state_d program"
where "Client =
  mk_total_program
  ({s. tok s ∈ atMost NbT &
    giv s = [] & ask s = [] & rel s = []},
  {rel_act, tok_act, ask_act},
  ⋃ G ∈ preserves rel Int preserves ask Int preserves tok.
  Acts G)"

definition
non_dummy :: "'a state_d => state"
where "non_dummy s = (|giv = giv s, ask = ask s, rel = rel s, tok = tok
s|)"

```

definition

```

client_map :: "'a state_d => state*'a"
where "client_map = funPair non_dummy dummy"

```

```

declare Client_def [THEN def_prg_Init, simp]
declare Client_def [THEN def_prg_AllowedActs, simp]
declare rel_act_def [THEN def_act_simp, simp]
declare tok_act_def [THEN def_act_simp, simp]
declare ask_act_def [THEN def_act_simp, simp]

```

```

lemma Client_ok_iff [iff]:
  "(Client ok G) =
   (G ∈ preserves rel & G ∈ preserves ask & G ∈ preserves tok &
    Client ∈ Allowed G)"
<proof>

```

Safety property 1: ask, rel are increasing

```

lemma increasing_ask_rel:
  "Client ∈ UNIV guarantees Increasing ask Int Increasing rel"
<proof>

```

```

declare nth_append [simp] append_one_prefix [simp]

```

Safety property 2: the client never requests too many tokens. With no Substitution Axiom, we must prove the two invariants simultaneously.

```

lemma ask_bounded_lemma:
  "Client ok G
   ==> Client ⊔ G ∈
     Always {s. tok s ≤ NbT} Int
     {s. ∀elt ∈ set (ask s). elt ≤ NbT}"
<proof>

```

export version, with no mention of tok in the postcondition, but unfortunately tok must be declared local.

```

lemma ask_bounded:
  "Client ∈ UNIV guarantees Always {s. ∀elt ∈ set (ask s). elt ≤ NbT}"
<proof>

```

**** Towards proving the liveness property ****

```

lemma stable_rel_le_giv: "Client ∈ stable {s. rel s ≤ giv s}"
<proof>

```

```

lemma Join_Stable_rel_le_giv:
  "[| Client ⊔ G ∈ Increasing giv; G ∈ preserves rel |]
   ==> Client ⊔ G ∈ Stable {s. rel s ≤ giv s}"
<proof>

```

```

lemma Join_Always_rel_le_giv:
  "[| Client ⊔ G ∈ Increasing giv; G ∈ preserves rel |]
   ==> Client ⊔ G ∈ Always {s. rel s ≤ giv s}"

```

<proof>

lemma *transient_lemma:*

"Client \in transient {s. rel s = k & k < h & h \leq giv s & h pfixGe ask s}"

<proof>

lemma *induct_lemma:*

"[| Client \sqcup G \in Increasing giv; Client ok G |]"

==> Client \sqcup G \in {s. rel s = k & k < h & h \leq giv s & h pfixGe ask s}

LeadsTo {s. k < rel s & rel s \leq giv s &

h \leq giv s & h pfixGe ask s}"

<proof>

lemma *rel_progress_lemma:*

"[| Client \sqcup G \in Increasing giv; Client ok G |]"

==> Client \sqcup G \in {s. rel s < h & h \leq giv s & h pfixGe ask s}

LeadsTo {s. h \leq rel s}"

<proof>

lemma *client_progress_lemma:*

"[| Client \sqcup G \in Increasing giv; Client ok G |]"

==> Client \sqcup G \in {s. h \leq giv s & h pfixGe ask s}

LeadsTo {s. h \leq rel s}"

<proof>

Progress property: all tokens that are given will be released

lemma *client_progress:*

"Client \in

Increasing giv guarantees

(INT h. {s. h \leq giv s & h pfixGe ask s} LeadsTo {s. h \leq rel s})"

<proof>

This shows that the Client won't alter other variables in any state that it is combined with

lemma *client_preserves_dummy:* "Client \in preserves dummy"

<proof>

* Obsolete lemmas from first version of the Client *

lemma *stable_size_rel_le_giv:*

"Client \in stable {s. size (rel s) \leq size (giv s)}"

<proof>

clients return the right number of tokens

lemma *ok_guar_rel_prefix_giv:*

"Client \in Increasing giv guarantees Always {s. rel s \leq giv s}"

<proof>

end

27 Projections of State Sets

theory *Project* imports *Extend* begin

definition *projecting* :: "['c program => 'c set, 'a*'b => 'c,
 'a program, 'c program set, 'a program set] => bool" where
 "projecting C h F X' X ==
 $\forall G. \text{extend } h \ F \sqcup G \in X' \ \longrightarrow \ F \sqcup \text{project } h \ (C \ G) \ G \in X$ "

definition *extending* :: "['c program => 'c set, 'a*'b => 'c, 'a program,
 'c program set, 'a program set] => bool" where
 "extending C h F Y' Y ==
 $\forall G. \text{extend } h \ F \ \text{ok } G \ \longrightarrow \ F \sqcup \text{project } h \ (C \ G) \ G \in Y$
 $\longrightarrow \ \text{extend } h \ F \sqcup G \in Y$ "

definition *subset_closed* :: "'a set set => bool" where
 "subset_closed U == $\forall A \in U. \text{Pow } A \subseteq U$ "

context *Extend*
 begin

lemma *project_extend_constrains_I*:
 "F \in A co B ==> project h C (extend h F) \in A co B"
 <proof>

27.1 Safety

lemma *project_unless*:
 "[| G \in stable C; project h C G \in A unless B |]
 ==> G \in (C \cap extend_set h A) unless (extend_set h B)"
 <proof>

lemma *Join_project_constrains*:
 "(F \sqcup project h C G \in A co B) =
 (extend h F \sqcup G \in (C \cap extend_set h A) co (extend_set h B) &
 F \in A co B)"
 <proof>

lemma *Join_project_stable*:
 "extend h F \sqcup G \in stable C
 ==> (F \sqcup project h C G \in stable A) =
 (extend h F \sqcup G \in stable (C \cap extend_set h A) &
 F \in stable A)"
 <proof>

lemma *project_constrains_I*:
 "extend h F \sqcup G \in extend_set h A co extend_set h B
 ==> F \sqcup project h C G \in A co B"
 <proof>

```

lemma project_increasing_I:
  "extend h F ⊔ G ∈ increasing (func o f)
   ==> F ⊔ project h C G ∈ increasing func"
<proof>

lemma Join_project_increasing:
  "(F ⊔ project h UNIV G ∈ increasing func) =
   (extend h F ⊔ G ∈ increasing (func o f))"
<proof>

lemma project_constrains_D:
  "F ⊔ project h UNIV G ∈ A co B
   ==> extend h F ⊔ G ∈ extend_set h A co extend_set h B"
<proof>

end

```

27.2 "projecting" and union/intersection (no converses)

```

lemma projecting_Int:
  "[| projecting C h F XA' XA; projecting C h F XB' XB |]
   ==> projecting C h F (XA' ∩ XB') (XA ∩ XB)"
<proof>

lemma projecting_Un:
  "[| projecting C h F XA' XA; projecting C h F XB' XB |]
   ==> projecting C h F (XA' ∪ XB') (XA ∪ XB)"
<proof>

lemma projecting_INT:
  "[| !!i. i ∈ I ==> projecting C h F (X' i) (X i) |]
   ==> projecting C h F (∩ i ∈ I. X' i) (∩ i ∈ I. X i)"
<proof>

lemma projecting_UN:
  "[| !!i. i ∈ I ==> projecting C h F (X' i) (X i) |]
   ==> projecting C h F (∪ i ∈ I. X' i) (∪ i ∈ I. X i)"
<proof>

lemma projecting_weaken:
  "[| projecting C h F X' X; U' <= X'; X ⊆ U |] ==> projecting C h F U'
  U"
<proof>

lemma projecting_weaken_L:
  "[| projecting C h F X' X; U' <= X' |] ==> projecting C h F U' X"
<proof>

lemma extending_Int:
  "[| extending C h F YA' YA; extending C h F YB' YB |]
   ==> extending C h F (YA' ∩ YB') (YA ∩ YB)"
<proof>

```

```

lemma extending_Un:
  "[| extending C h F YA' YA; extending C h F YB' YB |]
  ==> extending C h F (YA'  $\cup$  YB') (YA  $\cup$  YB)"
<proof>

lemma extending_INT:
  "[| !!i. i  $\in$  I ==> extending C h F (Y' i) (Y i) |]
  ==> extending C h F ( $\bigcap$  i  $\in$  I. Y' i) ( $\bigcap$  i  $\in$  I. Y i)"
<proof>

lemma extending_UN:
  "[| !!i. i  $\in$  I ==> extending C h F (Y' i) (Y i) |]
  ==> extending C h F ( $\bigcup$  i  $\in$  I. Y' i) ( $\bigcup$  i  $\in$  I. Y i)"
<proof>

lemma extending_weaken:
  "[| extending C h F Y' Y; Y' <= V'; V  $\subseteq$  Y |] ==> extending C h F V' V"
<proof>

lemma extending_weaken_L:
  "[| extending C h F Y' Y; Y' <= V' |] ==> extending C h F V' Y"
<proof>

lemma projecting_UNIV: "projecting C h F X' UNIV"
<proof>

context Extend
begin

lemma projecting_constrains:
  "projecting C h F (extend_set h A co extend_set h B) (A co B)"
<proof>

lemma projecting_stable:
  "projecting C h F (stable (extend_set h A)) (stable A)"
<proof>

lemma projecting_increasing:
  "projecting C h F (increasing (func o f)) (increasing func)"
<proof>

lemma extending_UNIV: "extending C h F UNIV Y"
<proof>

lemma extending_constrains:
  "extending (%G. UNIV) h F (extend_set h A co extend_set h B) (A co B)"
<proof>

lemma extending_stable:
  "extending (%G. UNIV) h F (stable (extend_set h A)) (stable A)"
<proof>

lemma extending_increasing:
  "extending (%G. UNIV) h F (increasing (func o f)) (increasing func)"

```


<proof>

27.3 Reachability and project

lemma *reachable_imp_reachable_project*:
 "[| reachable (extend h F⊔G) ⊆ C;
 z ∈ reachable (extend h F⊔G) |]
 ==> f z ∈ reachable (F⊔project h C G)"

<proof>

lemma *project_Constrains_D*:
 "F⊔project h (reachable (extend h F⊔G)) G ∈ A Co B
 ==> extend h F⊔G ∈ (extend_set h A) Co (extend_set h B)"

<proof>

lemma *project_Stable_D*:
 "F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A
 ==> extend h F⊔G ∈ Stable (extend_set h A)"

<proof>

lemma *project_Always_D*:
 "F⊔project h (reachable (extend h F⊔G)) G ∈ Always A
 ==> extend h F⊔G ∈ Always (extend_set h A)"

<proof>

lemma *project_Increasing_D*:
 "F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func
 ==> extend h F⊔G ∈ Increasing (func o f)"

<proof>

27.4 Converse results for weak safety: benefits of the argument C

lemma *reachable_project_imp_reachable*:
 "[| C ⊆ reachable(extend h F⊔G);
 x ∈ reachable (F⊔project h C G) |]
 ==> ∃y. h(x,y) ∈ reachable (extend h F⊔G)"

<proof>

lemma *project_set_reachable_extend_eq*:
 "project_set h (reachable (extend h F⊔G)) =
 reachable (F⊔project h (reachable (extend h F⊔G)) G)"

<proof>

lemma *reachable_extend_Join_subset*:
 "reachable (extend h F⊔G) ⊆ C
 ==> reachable (extend h F⊔G) ⊆
 extend_set h (reachable (F⊔project h C G))"

<proof>

lemma *project_Constrains_I*:
 "extend h F⊔G ∈ (extend_set h A) Co (extend_set h B)
 ==> F⊔project h (reachable (extend h F⊔G)) G ∈ A Co B"

<proof>

lemma *project_Stable_I:*

"extend h F⊔G ∈ Stable (extend_set h A)
=> F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A"

<proof>

lemma *project_Always_I:*

"extend h F⊔G ∈ Always (extend_set h A)
=> F⊔project h (reachable (extend h F⊔G)) G ∈ Always A"

<proof>

lemma *project_Increasing_I:*

"extend h F⊔G ∈ Increasing (func o f)
=> F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func"

<proof>

lemma *project_Constrains:*

"(F⊔project h (reachable (extend h F⊔G)) G ∈ A Co B) =
(extend h F⊔G ∈ (extend_set h A) Co (extend_set h B))"

<proof>

lemma *project_Stable:*

"(F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A) =
(extend h F⊔G ∈ Stable (extend_set h A))"

<proof>

lemma *project_Increasing:*

"(F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func) =
(extend h F⊔G ∈ Increasing (func o f))"

<proof>

27.5 A lot of redundant theorems: all are proved to facilitate reasoning about guarantees.

lemma *projecting_Constrains:*

"projecting (%G. reachable (extend h F⊔G)) h F
(extend_set h A Co extend_set h B) (A Co B)"

<proof>

lemma *projecting_Stable:*

"projecting (%G. reachable (extend h F⊔G)) h F
(Stable (extend_set h A)) (Stable A)"

<proof>

lemma *projecting_Always:*

"projecting (%G. reachable (extend h F⊔G)) h F
(Always (extend_set h A)) (Always A)"

<proof>

lemma *projecting_Increasing:*

"projecting (%G. reachable (extend h F⊔G)) h F
(Increasing (func o f)) (Increasing func)"

<proof>

lemma *extending_Constrains:*

```
"extending (%G. reachable (extend h F⊔G)) h F
  (extend_set h A Co extend_set h B) (A Co B)"
```

<proof>

lemma *extending_Stable:*

```
"extending (%G. reachable (extend h F⊔G)) h F
  (Stable (extend_set h A)) (Stable A)"
```

<proof>

lemma *extending_Always:*

```
"extending (%G. reachable (extend h F⊔G)) h F
  (Always (extend_set h A)) (Always A)"
```

<proof>

lemma *extending_Increasing:*

```
"extending (%G. reachable (extend h F⊔G)) h F
  (Increasing (func o f)) (Increasing func)"
```

<proof>

27.6 leadsETo in the precondition (??)

27.6.1 transient

lemma *transient_extend_set_imp_project_transient:*

```
"[| G ∈ transient (C ∩ extend_set h A); G ∈ stable C |]
  ==> project h C G ∈ transient (project_set h C ∩ A)"
```

<proof>

lemma *project_extend_transient_D:*

```
"project h C (extend h F) ∈ transient (project_set h C ∩ D)
  ==> F ∈ transient (project_set h C ∩ D)"
```

<proof>

27.6.2 ensures – a primitive combining progress with safety

lemma *ensures_extend_set_imp_project_ensures:*

```
"[| extend h F ∈ stable C; G ∈ stable C;
  extend h F⊔G ∈ A ensures B; A-B = C ∩ extend_set h D |]
  ==> F⊔project h C G
  ∈ (project_set h C ∩ project_set h A) ensures (project_set h B)"
```

<proof>

Transferring a transient property upwards

lemma *project_transient_extend_set:*

```
"project h C G ∈ transient (project_set h C ∩ A - B)
  ==> G ∈ transient (C ∩ extend_set h A - extend_set h B)"
```

<proof>

lemma *project_unless2:*

```
"[| G ∈ stable C; project h C G ∈ (project_set h C ∩ A) unless B |]"
```

$\implies G \in (C \cap \text{extend_set } h A) \text{ unless } (\text{extend_set } h B)$ "
 <proof>

lemma *extend_unless*:
 "[| extend h F \in stable C; F \in A unless B |]"
 $\implies \text{extend } h F \in C \cap \text{extend_set } h A \text{ unless } \text{extend_set } h B$ "
 <proof>

lemma *Join_project_ensures*:
 "[| extend h F \sqcup G \in stable C;
 F \sqcup project h C G \in A ensures B |]"
 $\implies \text{extend } h F \sqcup G \in (C \cap \text{extend_set } h A) \text{ ensures } (\text{extend_set } h B)$ "
 <proof>

Lemma useful for both STRONG and WEAK progress, but the transient condition's very strong

lemma *PLD_lemma*:
 "[| extend h F \sqcup G \in stable C;
 F \sqcup project h C G \in (project_set h C \cap A) leadsTo B |]"
 $\implies \text{extend } h F \sqcup G \in C \cap \text{extend_set } h (\text{project_set } h C \cap A) \text{ leadsTo } (\text{extend_set } h B)$ "
 <proof>

lemma *project_leadsTo_D_lemma*:
 "[| extend h F \sqcup G \in stable C;
 F \sqcup project h C G \in (project_set h C \cap A) leadsTo B |]"
 $\implies \text{extend } h F \sqcup G \in (C \cap \text{extend_set } h A) \text{ leadsTo } (\text{extend_set } h B)$ "
 <proof>

lemma *Join_project_LeadsTo*:
 "[| C = (reachable (extend h F \sqcup G));
 F \sqcup project h C G \in A LeadsTo B |]"
 $\implies \text{extend } h F \sqcup G \in (\text{extend_set } h A) \text{ LeadsTo } (\text{extend_set } h B)$ "
 <proof>

27.7 Towards the theorem *project_Ensures_D*

lemma *project_ensures_D_lemma*:
 "[| G \in stable ((C \cap extend_set h A) - (extend_set h B));
 F \sqcup project h C G \in (project_set h C \cap A) ensures B;
 extend h F \sqcup G \in stable C |]"
 $\implies \text{extend } h F \sqcup G \in (C \cap \text{extend_set } h A) \text{ ensures } (\text{extend_set } h B)$ "
 <proof>

lemma *project_ensures_D*:
 "[| F \sqcup project h UNIV G \in A ensures B;
 G \in stable (extend_set h A - extend_set h B) |]"
 $\implies \text{extend } h F \sqcup G \in (\text{extend_set } h A) \text{ ensures } (\text{extend_set } h B)$ "
 <proof>

lemma *project_Ensures_D*:

```

    "[| F⊔project h (reachable (extend h F⊔G)) G ∈ A Ensures B;
      G ∈ stable (reachable (extend h F⊔G) ∩ extend_set h A -
        extend_set h B) |]
    ==> extend h F⊔G ∈ (extend_set h A) Ensures (extend_set h B)"
  <proof>

```

27.8 Guarantees

```

lemma project_act.Restrict_subset_project_act:
  "project_act h (Restrict C act) ⊆ project_act h act"
  <proof>

```

```

lemma subset_closed_ok_extend_imp_ok_project:
  "[| extend h F ok G; subset_closed (AllowedActs F) |]
  ==> F ok project h C G"
  <proof>

```

```

lemma project_guarantees_raw:
  assumes xguary: "F ∈ X guarantees Y"
  and closed: "subset_closed (AllowedActs F)"
  and project: "!!G. extend h F⊔G ∈ X'
    ==> F⊔project h (C G) G ∈ X"
  and extend: "!!G. [| F⊔project h (C G) G ∈ Y |]
    ==> extend h F⊔G ∈ Y'"
  shows "extend h F ∈ X' guarantees Y'"
  <proof>

```

```

lemma project_guarantees:
  "[| F ∈ X guarantees Y; subset_closed (AllowedActs F);
    projecting C h F X' X; extending C h F Y' Y |]
  ==> extend h F ∈ X' guarantees Y'"
  <proof>

```

27.9 guarantees corollaries

27.9.1 Some could be deleted: the required versions are easy to prove

```

lemma extend_guar_increasing:
  "[| F ∈ UNIV guarantees increasing func;
    subset_closed (AllowedActs F) |]
  ==> extend h F ∈ X' guarantees increasing (func o f)"
  <proof>

```

```

lemma extend_guar_Increasing:
  "[| F ∈ UNIV guarantees Increasing func;
    subset_closed (AllowedActs F) |]
  ==> extend h F ∈ X' guarantees Increasing (func o f)"
  <proof>

```

```

lemma extend_guar_Always:

```

```

    "[| F ∈ Always A guarantees Always B;
      subset_closed (AllowedActs F) |]
    ==> extend h F
      ∈ Always(extend_set h A) guarantees Always(extend_set h B)"
  <proof>

```

27.9.2 Guarantees with a leadsTo postcondition

```

lemma project_leadsTo_D:
  "F ⊔ project h UNIV G ∈ A leadsTo B
  ==> extend h F ⊔ G ∈ (extend_set h A) leadsTo (extend_set h B)"
  <proof>

```

```

lemma project_LeadsTo_D:
  "F ⊔ project h (reachable (extend h F ⊔ G)) G ∈ A LeadsTo B
  ==> extend h F ⊔ G ∈ (extend_set h A) LeadsTo (extend_set h B)"
  <proof>

```

```

lemma extending_leadsTo:
  "extending (%G. UNIV) h F
   (extend_set h A leadsTo extend_set h B) (A leadsTo B)"
  <proof>

```

```

lemma extending_LeadsTo:
  "extending (%G. reachable (extend h F ⊔ G)) h F
   (extend_set h A LeadsTo extend_set h B) (A LeadsTo B)"
  <proof>

```

end

end

28 Progress Under Allowable Sets

theory *ELT* imports *Project* begin

inductive_set

```

  elt :: "[ 'a set set, 'a program ] => ( 'a set * 'a set ) set"
  for CC :: "'a set set" and F :: "'a program"
  where

    Basis: "[| F ∈ A ensures B; A-B ∈ (insert {} CC) |] ==> (A,B) ∈ elt
    CC F"

    | Trans: "[| (A,B) ∈ elt CC F; (B,C) ∈ elt CC F |] ==> (A,C) ∈ elt CC F"

    | Union: "∀ A ∈ S. (A,B) ∈ elt CC F ==> (Union S, B) ∈ elt CC F"

```

definition

```

  givenBy :: "[ 'a => 'b ] => 'a set set"
  where "givenBy f = range (%B. f-` B)"

```

definition

```

leadsETo :: "[ 'a set, 'a set set, 'a set ] => 'a program set"
           ("(3_/ leadsTo[_]/ _)" [80,0,80] 80)
where "leadsETo A CC B = {F. (A,B) ∈ elt CC F}"

```

definition

```

LeadsETo :: "[ 'a set, 'a set set, 'a set ] => 'a program set"
           ("(3_/ LeadsTo[_]/ _)" [80,0,80] 80)
where "LeadsETo A CC B =
      {F. F ∈ (reachable F Int A) leadsTo[(%C. reachable F Int C) ' CC] B}"

```

lemma givenBy_id [simp]: "givenBy id = UNIV"

<proof>

lemma givenBy_eq_all: "(givenBy v) = {A. $\forall x \in A. \forall y. v x = v y \longrightarrow y \in A$ }"

<proof>

lemma givenByI: " $(\bigwedge x y. [x \in A; v x = v y] \implies y \in A) \implies A \in \text{givenBy } v$ "

<proof>

lemma givenByD: "[A ∈ givenBy v; x ∈ A; v x = v y] $\implies y \in A$ "

<proof>

lemma empty_mem_givenBy [iff]: "{ } ∈ givenBy v"

<proof>

lemma givenBy_imp_eq_Collect: "A ∈ givenBy v $\implies \exists P. A = \{s. P(v s)\}$ "

<proof>

lemma Collect_mem_givenBy: "{s. P(v s)} ∈ givenBy v"

<proof>

lemma givenBy_eq_Collect: "givenBy v = {A. $\exists P. A = \{s. P(v s)\}}$ "

<proof>

lemma preserves_givenBy_imp_stable:

"[F ∈ preserves v; D ∈ givenBy v] $\implies F \in \text{stable } D$ "

<proof>

lemma givenBy_o_subset: "givenBy (w o v) \leq givenBy v"

<proof>

lemma givenBy_DiffI:

"[A ∈ givenBy v; B ∈ givenBy v] $\implies A - B \in \text{givenBy } v$ "

<proof>

```

lemma leadsETo_Basis [intro]:
  "[| F ∈ A ensures B; A-B ∈ insert {} CC |] ==> F ∈ A leadsTo[CC] B"
  <proof>

lemma leadsETo_Trans:
  "[| F ∈ A leadsTo[CC] B; F ∈ B leadsTo[CC] C |] ==> F ∈ A leadsTo[CC]
  C"
  <proof>

lemma leadsETo_Un_duplicate:
  "F ∈ A leadsTo[CC] (A' ∪ A') ==> F ∈ A leadsTo[CC] A'"
  <proof>

lemma leadsETo_Un_duplicate2:
  "F ∈ A leadsTo[CC] (A' ∪ C ∪ C) ==> F ∈ A leadsTo[CC] (A' Un C)"
  <proof>

lemma leadsETo_Union:
  "(∧ A. A ∈ S ==> F ∈ A leadsTo[CC] B) ==> F ∈ (∪ S) leadsTo[CC] B"
  <proof>

lemma leadsETo_UN:
  "(∧ i. i ∈ I ==> F ∈ (A i) leadsTo[CC] B)
  ==> F ∈ (UN i:I. A i) leadsTo[CC] B"
  <proof>

lemma leadsETo_induct:
  "[| F ∈ za leadsTo[CC] zb;
  !!A B. [| F ∈ A ensures B; A-B ∈ insert {} CC |] ==> P A B;
  !!A B C. [| F ∈ A leadsTo[CC] B; P A B; F ∈ B leadsTo[CC] C; P B C
  |]
  ==> P A C;
  !!B S. ∀ A ∈ S. F ∈ A leadsTo[CC] B & P A B ==> P (∪ S) B
  |] ==> P za zb"
  <proof>

lemma leadsETo_mono: "CC' <= CC ==> (A leadsTo[CC'] B) <= (A leadsTo[CC]
  B)"
  <proof>

lemma leadsETo_Trans_Un:
  "[| F ∈ A leadsTo[CC] B; F ∈ B leadsTo[DD] C |]
  ==> F ∈ A leadsTo[CC Un DD] C"
  <proof>

```


lemma *leadsETo_Union_Int*:
 "(!!A. A ∈ S ==> F ∈ (A Int C) leadsTo[CC] B)
 ==> F ∈ (⋃ S Int C) leadsTo[CC] B"
 ⟨proof⟩

lemma *leadsETo_Un*:
 "[| F ∈ A leadsTo[CC] C; F ∈ B leadsTo[CC] C |]
 ==> F ∈ (A Un B) leadsTo[CC] C"
 ⟨proof⟩

lemma *single_leadsETo_I*:
 "(⋀x. x ∈ A ==> F ∈ {x} leadsTo[CC] B) ==> F ∈ A leadsTo[CC] B"
 ⟨proof⟩

lemma *subset_imp_leadsETo*: "A<=B ==> F ∈ A leadsTo[CC] B"
 ⟨proof⟩

lemmas *empty_leadsETo = empty_subsetI [THEN subset_imp_leadsETo, simp]*

lemma *leadsETo_weaken_R*:
 "[| F ∈ A leadsTo[CC] A'; A'<=B' |] ==> F ∈ A leadsTo[CC] B'"
 ⟨proof⟩

lemma *leadsETo_weaken_L*:
 "[| F ∈ A leadsTo[CC] A'; B<=A |] ==> F ∈ B leadsTo[CC] A'"
 ⟨proof⟩

lemma *leadsETo_Un_distrib*:
 "F ∈ (A Un B) leadsTo[CC] C =
 (F ∈ A leadsTo[CC] C ∧ F ∈ B leadsTo[CC] C)"
 ⟨proof⟩

lemma *leadsETo_UN_distrib*:
 "F ∈ (UN i:I. A i) leadsTo[CC] B =
 (∀i∈I. F ∈ (A i) leadsTo[CC] B)"
 ⟨proof⟩

lemma *leadsETo_Union_distrib*:
 "F ∈ (⋃ S) leadsTo[CC] B = (∀A∈S. F ∈ A leadsTo[CC] B)"
 ⟨proof⟩

lemma *leadsETo_weaken*:
 "[| F ∈ A leadsTo[CC'] A'; B<=A; A'<=B'; CC' <= CC |]
 ==> F ∈ B leadsTo[CC] B'"
 ⟨proof⟩

lemma *leadsETo_givenBy*:

```

    "[| F ∈ A leadsTo[CC] A'; CC ≤ givenBy v |]
    ==> F ∈ A leadsTo[givenBy v] A'"
  <proof>

```

```

lemma leadsETo_Diff:
  "[| F ∈ (A-B) leadsTo[CC] C; F ∈ B leadsTo[CC] C |]
  ==> F ∈ A leadsTo[CC] C"
  <proof>

```

```

lemma leadsETo_Un_Un:
  "[| F ∈ A leadsTo[CC] A'; F ∈ B leadsTo[CC] B' |]
  ==> F ∈ (A Un B) leadsTo[CC] (A' Un B)'"
  <proof>

```

```

lemma leadsETo_cancel2:
  "[| F ∈ A leadsTo[CC] (A' Un B); F ∈ B leadsTo[CC] B' |]
  ==> F ∈ A leadsTo[CC] (A' Un B)'"
  <proof>

```

```

lemma leadsETo_cancel1:
  "[| F ∈ A leadsTo[CC] (B Un A'); F ∈ B leadsTo[CC] B' |]
  ==> F ∈ A leadsTo[CC] (B' Un A)'"
  <proof>

```

```

lemma leadsETo_cancel_Diff1:
  "[| F ∈ A leadsTo[CC] (B Un A'); F ∈ (B-A') leadsTo[CC] B' |]
  ==> F ∈ A leadsTo[CC] (B' Un A)'"
  <proof>

```

```

lemma e_psp_stable:
  "[| F ∈ A leadsTo[CC] A'; F ∈ stable B; ∀ C ∈ CC. C Int B ∈ CC |]
  ==> F ∈ (A Int B) leadsTo[CC] (A' Int B)'"
  <proof>

```

```

lemma e_psp_stable2:
  "[| F ∈ A leadsTo[CC] A'; F ∈ stable B; ∀ C ∈ CC. C Int B ∈ CC |]
  ==> F ∈ (B Int A) leadsTo[CC] (B Int A)'"
  <proof>

```

```

lemma e_psp:
  "[| F ∈ A leadsTo[CC] A'; F ∈ B co B';
  ∀ C ∈ CC. C Int B Int B' ∈ CC |]
  ==> F ∈ (A Int B') leadsTo[CC] ((A' Int B) Un (B' - B))'"

```

<proof>

lemma e_psp2:

```
"[| F ∈ A leadsTo[CC] A'; F ∈ B co B';
  ∀ C ∈ CC. C Int B Int B' ∈ CC |]
  ==> F ∈ (B' Int A) leadsTo[CC] ((B Int A') Un (B' - B))"
```

<proof>

lemma gen_leadsETo_imp_Join_leadsETo:

```
"[| F ∈ (A leadsTo[givenBy v] B); G ∈ preserves v;
  F ⊔ G ∈ stable C |]
  ==> F ⊔ G ∈ ((C Int A) leadsTo[(%D. C Int D) ' givenBy v] B)"
```

<proof>

lemma leadsETo_subset_leadsTo: "(A leadsTo[CC] B) <= (A leadsTo B)"

<proof>

lemma leadsETo_UNIV_eq_leadsTo: "(A leadsTo[UNIV] B) = (A leadsTo B)"

<proof>

lemma LeadsETo_eq_leadsETo:

```
"A LeadsTo[CC] B =
  {F. F ∈ (reachable F Int A) leadsTo[(%C. reachable F Int C) ' CC]
    (reachable F Int B)}"
```

<proof>

lemma LeadsETo_Trans:

```
"[| F ∈ A LeadsTo[CC] B; F ∈ B LeadsTo[CC] C |]
  ==> F ∈ A LeadsTo[CC] C"
```

<proof>

lemma LeadsETo_Union:

```
"(∧ A. A ∈ S ⇒ F ∈ A LeadsTo[CC] B) ⇒ F ∈ (∪ S) LeadsTo[CC] B"
```

<proof>

lemma LeadsETo_UN:

```
"(∧ i. i ∈ I ⇒ F ∈ (A i) LeadsTo[CC] B)
  ⇒ F ∈ (UN i:I. A i) LeadsTo[CC] B"
```

<proof>

lemma LeadsETo_Un:

"[| F ∈ A LeadsTo[CC] C; F ∈ B LeadsTo[CC] C |]
 ==> F ∈ (A Un B) LeadsTo[CC] C"
 <proof>

lemma single_LeadsETo_I:

"(∧ s. s ∈ A ==> F ∈ {s} LeadsTo[CC] B) ==> F ∈ A LeadsTo[CC] B"
 <proof>

lemma subset_imp_LeadsETo:

"A <= B ==> F ∈ A LeadsTo[CC] B"
 <proof>

lemmas empty_LeadsETo = empty_subsetI [THEN subset_imp_LeadsETo]

lemma LeadsETo_weaken_R:

"[| F ∈ A LeadsTo[CC] A'; A' <= B' |] ==> F ∈ A LeadsTo[CC] B'"
 <proof>

lemma LeadsETo_weaken_L:

"[| F ∈ A LeadsTo[CC] A'; B <= A |] ==> F ∈ B LeadsTo[CC] A'"
 <proof>

lemma LeadsETo_weaken:

"[| F ∈ A LeadsTo[CC'] A';
 B <= A; A' <= B'; CC' <= CC |]
 ==> F ∈ B LeadsTo[CC] B'"
 <proof>

lemma LeadsETo_subset_LeadsTo: "(A LeadsTo[CC] B) <= (A LeadsTo B)"

<proof>

lemma reachable_ensures:

"F ∈ A ensures B ==> F ∈ (reachable F Int A) ensures B"
 <proof>

lemma le1_lemma:

"F ∈ A leadsTo B ==> F ∈ (reachable F Int A) leadsTo[Pow(reachable F)]
 B"
 <proof>

lemma LeadsETo_UNIV_eq_LeadsTo: "(A LeadsTo[UNIV] B) = (A LeadsTo B)"

<proof>

context Extend

begin

lemma givenBy_o_eq_extend_set:

"givenBy (v o f) = extend_set h ' (givenBy v)"

<proof>

lemma *givenBy_eq_extend_set*: "givenBy f = range (extend_set h)"

<proof>

lemma *extend_set_givenBy_I*:

"D ∈ givenBy v ==> extend_set h D ∈ givenBy (v o f)"

<proof>

lemma *leadsETo_imp_extend_leadsETo*:

"F ∈ A leadsTo[CC] B

==> extend h F ∈ (extend_set h A) leadsTo[extend_set h ' CC]
(extend_set h B)"

<proof>

lemma *Join_project_ensures_strong*:

"[| project h C G ∉ transient (project_set h C Int (A-B)) |
project_set h C Int (A - B) = {}];

extend h F ⊔ G ∈ stable C;

F ⊔ project h C G ∈ (project_set h C Int A) ensures B |]

==> extend h F ⊔ G ∈ (C Int extend_set h A) ensures (extend_set h B)"

<proof>

lemma *pli_lemma*:

"[| extend h F ⊔ G ∈ stable C;

F ⊔ project h C G

∈ project_set h C Int project_set h A leadsTo project_set h B |]

==> F ⊔ project h C G

∈ project_set h C Int project_set h A leadsTo

project_set h C Int project_set h B"

<proof>

lemma *project_leadsETo_I_lemma*:

"[| extend h F ⊔ G ∈ stable C;

extend h F ⊔ G ∈

(C Int A) leadsTo[(%D. C Int D)'givenBy f] B |]

==> F ⊔ project h C G

∈ (project_set h C Int project_set h (C Int A)) leadsTo (project_set h
B)"

<proof>

lemma *project_leadsETo_I*:

"extend h F ⊔ G ∈ (extend_set h A) leadsTo[givenBy f] (extend_set h B)

==> F ⊔ project h UNIV G ∈ A leadsTo B"

<proof>

```

lemma project_LeadsETo_I:
  "extend h F ⊔ G ∈ (extend_set h A) LeadsTo[givenBy f] (extend_set h B)

  ⇒ F ⊔ project h (reachable (extend h F ⊔ G)) G
  ∈ A LeadsTo B"
<proof>

```

```

lemma projecting_leadsTo:
  "projecting (λG. UNIV) h F
  (extend_set h A leadsTo[givenBy f] extend_set h B)
  (A leadsTo B)"
<proof>

```

```

lemma projecting_LeadsTo:
  "projecting (λG. reachable (extend h F ⊔ G)) h F
  (extend_set h A LeadsTo[givenBy f] extend_set h B)
  (A LeadsTo B)"
<proof>

```

end

end